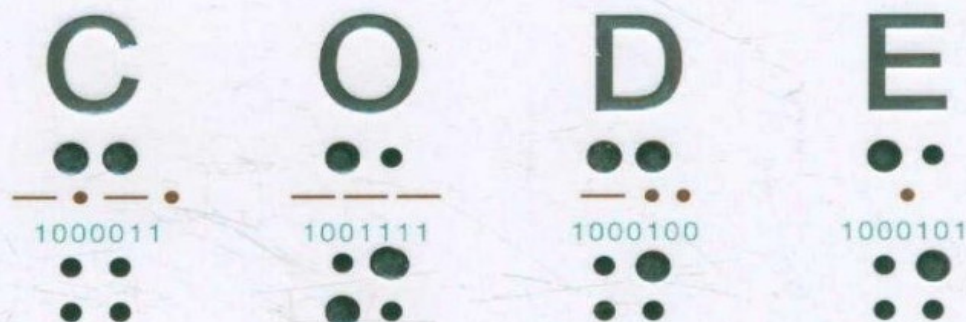


100001110011111000100100010110000111001111100010010001011000011100111110001001000101

永不褪色的计算机科学经典著作



编 码

隐匿在计算机软硬件背后的语言

Code: The Hidden Language of
Computer Hardware and Software

[美] Charles Petzold 著
左飞 薛佟佟 译

第 1 章 电筒密谈

假若你才 10 岁，你的好朋友与你临街而住，而且你们卧室的窗户面对着面。每天晚上，当父母像平常一样很早催你上床睡觉时，你可能还想与好朋友交流思想、发现、小秘密、传闻、笑话和梦想，没有人可以责备你，毕竟，渴望交流是大多数人的天性。

当你们卧室还亮着灯时，你和你的好朋友可以临窗舞动手臂、打手势或以身体语言来交流思想，但复杂一些的交流就有些困难了。而且一旦父母宣布“熄灯”，交流也就无法继续进行了。

如何联系呢？用电话吗？10 岁的小孩子屋里有电话吗？即使有，你们的谈话可能被偷听。如果家里的电脑通过电话线联了网，它可能会提供无声的帮助，不过很不幸，它也不会你的房间里。

你和朋友采用的方法是用手电筒。所有的人都知道手电筒是为孩子们藏在被窝里看书而发明的，它也适合在黑暗中用来交流。它无声无息，且光的方向性很好，不会从卧室的门缝中泄露而使家人起疑。

用手电筒的光可以交谈吗？这值得一试。一年级你就学过在纸上写字母和单词，把这种方法运用到手电筒上看起来也合情合理。你所需做的就是临窗而站，用光画出字母。画字母‘O’，就打开电筒，在空中画个圈，然后关上开关；字母‘I’则是画竖直的一笔。但是你很快发现这种方法行不通，当你注视来去飞舞的光柱

时，会发现在脑海中将它们组合起来不是件容易的事，这些光划成的圈圈杠杠太不准确了。

也许你曾经看过一部电影，影片中两个水手隔海用闪烁的光传递消息。在另一部电影中，一个间谍用镜子反射阳光向一间屋子中被俘获的同伙发送讯息。这就给了你启发，你起先设计一种简单的交流方法，使字母表中的每个字母与一定数目的闪烁相对应。A闪一下，B闪两下，C闪三下，如此递推，Z就闪烁26下。

BAD这个词由字母间有间隔的两闪、一闪、四闪组成，这样你不会误以为它是闪七下的字母G了。词间的停顿则比字母间的停顿时间稍长一些。

这看起来很有希望，采用这种方法的优点是你不需要在空中挥舞手电筒，只需对准方向按开关就行了；缺点是你试图发送的第一个消息（“**How are you?**”）就需要131次闪烁，更糟的是，你忘了定义标点符号，所以无法表示句尾的问号了。

这离问题的解决已经很近了，你想别人以前肯定也遇到过类似的问题，你解决它的思想一定是正确的。为了解决问题，白天的图书馆之行使你发现了神奇的摩尔斯电码（**morse code**），这正是你想要的，即使你不得不重新学习如何“写”字母表中的字母。

以下就是区别：在你发明的体系中，每个字母是一定数目的闪烁，从闪烁一下的A到闪烁26的Z；而在摩尔斯电码中，有长短两种闪烁，当然，这会使摩尔斯电码更为复杂，但它在实际应用中却被证实是更有效的。那句“**How are you?**”现在仅需32次而不是131次闪烁，而且这还包含了问号。

在讨论摩尔斯电码的工作原理时，人们并不说“长闪烁”、“短闪烁”，他们使用“点

（**dot**）”和“划（**dash**）”，因为这样易于在印刷品上表示。在摩尔斯电码中，字母表中的每一

A	· · · · ·	J	· · · · · · · · · · ·	S	· · · · ·
B	· · · · · · · · · ·	K	· · · · · · · · ·	T	· · · · ·
C	· · · · · · · · · · ·	L	· · · · · · · · ·	U	· · · · · · · · ·
D	· · · · · · · · ·	M	· · · · · · · · ·	V	· · · · · · · · · ·
E	· · · · ·	N	· · · · · · · · ·	W	· · · · · · · · · ·
F	· · · · · · · · · ·	O	· · · · · · · · · · ·	X	· · · · · · · · · · ·
G	· · · · · · · · · ·	P	· · · · · · · · · · ·	Y	· · · · · · · · · · ·
H	· · · · · · · · ·	Q	· · · · · · · · · · ·	Z	· · · · · · · · · · ·
I	· · · · ·	R	· · · · · · · · · ·		

个字母与一个点划序列相对应，正如在下表中你所看到的：

尽管摩尔斯电码与计算机毫不相关，但熟悉它的本质却对深入了解计算机内部语言和软硬件的内部结构有很大的帮助。

在本书中，编码或代码（**code**）通常指一种在人和机器之间进行信息转换的系统（体系）。换句话说，编码便是交流。有时我们将编码看成是密码(机密)，其实大多数编码并不是的。大多数的编码都需要被很好地理解，因为它们是人类交流的基础。

在《百年孤独》的一书的开篇，马尔克斯回忆了一个时代，那时“世界一片混沌，许多事物没有名字。为了加以区别才给事物各个命名。”这些名字都是随意的，没有什么原因说明为什么不把猫称为狗或不把狗称为猫。可以说英语词汇就是一种编码。

我们用嘴发出声音组成单词，这些词可以为那些听得到我们声音，理解我们所用语言的人所听懂，我们称这种编码为“口头语言”或“语音”。对写在纸上（或凿在石头上、刻在木头上或通过比划写在空气中）的词，还有一种编码方式，那就是我们在印刷的报刊，杂志和书籍上看到的字符，称之为“书面语言”或“文本”。在许多语言中，语音和文本间有很强的联系。例如在英语中，字母或一组字母与一定的读音相对应。

手势语言的发明帮助了聋哑人进行面对面的交流。这是一种用手和胳膊的动作组合来表达词语中的单个字母、整个词及其基本概念的语言。对盲人来说，他们可以使用布莱叶盲文

(Braille)。这种文字使用凸起的点代表字母，字母串和单词。当谈话内容要被迅速地记录下来时，缩写和速记是很有用的。

人们在相互沟通时使用了各种不同的编码，因为在不同的应用场合，其中的一些较其他的更为简便。例如，语言不能在纸上存储，所以使用了文字；语言、文字不适合用来在黑夜中安静地传递消息，故摩尔斯电码是一个方便的替代品。只要一种编码可以适用于其他编码所不能适用的场合，它就是一种有用的编码。

以后将看到，计算机中使用了不同的编码来传递和存储数字、声音、音乐、图像和视频

(电影)。计算机不能直接处理人类世界的编码，因为它不能模拟人类的眼睛、鼻子、嘴和手指来接收信息。尽管这些年来计算机的发展趋势使我们的桌上电脑具有捕获、存储、处理和提供人类交流中所使用的各种信息的能，而且不论这些信息是视觉的(文字和图片)、听觉的

(语言、声音及音乐)还是两者的混合(动画和电影)。所有这些信
息都要求使用它们自己的编码方式，正如交谈需要使用人的某些器官(嘴和耳朵)，而书写和阅读则需要使用另外一些

器官（手和眼睛）一样。用手电筒发送摩尔斯电码时，电筒的开关快速地合开代表一个点，让电筒照亮稍长的时

间则代表一个划。举例来说，发送字母 A，要先快速地合开开关，然后再稍慢些合开。在发送下一个字母前要有短暂的停顿。约定划的时间大约是点的 3 倍。例如，如果点的照亮时间为 1 秒，那么划就是 3 秒。（实际上，摩尔斯电码的传递速度要快得多。）接收者看到了短闪和长闪 就知道是 A。

摩尔斯电码中点划之间的间隔是极为关键的。例如，发送字母 A 时，点划之间的间隔应与一个点的时间大致相同（如果点的时间是 1 秒，那么间隔的时间也是 1 秒）。同一个词中字母间 间隔稍长，约为划的持续时间（或者 3 秒，如果那是划的持续时间的 话）。下面是单词“hello”对应的摩尔斯电码，图中示意了字母间的间隔（隙）：

单词之间相隔大约 2 倍于划的时间（如果划是 3 秒，那么间隔即为 6 秒）。下面是“hi there”对应的编码（码字）：

手电筒开和关的时间长度并没有限定，这取决于点的时间长度，点长又由手电筒开关触发的速度和摩尔斯电码发送者记忆电码的熟练程度来决定，熟练发送者的划也许与生手的点等长。这个小问题会使接收电码有些困难，但在一两个字母之后，接收者通常就可以辨认出哪个是点，哪个是划了。

粗看起来，摩尔斯电码的定义——这里所谓的定义是指与字母表中的字母相对应的各种点划序列——与打字机字母的排列一样是随意的。但仔细观察后你会发现不完全如此，简短的码字分配给了使用频率较高的字母，例如E和T，爱赌博的人和“财富之轮”爱好者可能一下就注意到了这个问题；不常用的字母如Q和Z（它们在赌局中是10点）则分配以较长的码字。

几乎所有人都知道一点儿摩尔斯电码，国际遇险信号SOS的摩尔斯电码为“三点三划三点”。SOS并非缩写，选择它仅仅因为它有一个易记的摩尔斯电码序列。第二次世界大战中，英国广播公司选用贝多芬第五交响曲中的片段作为节目前奏——BAH、BAH、BAH、BAHMMMM，听起来颇像摩尔斯电码中V（代表Victory）的码字。

摩尔斯电码的一个缺点是它没有对大小写字母进行区分。除表示字母外，摩尔斯电码还用5位长的码字来表示数字：

1	· — — — —	6	— — — — ·
2	· · — — —	7	— — — — —
3	· · · — —	8	— — — — —
4	· · · · —	9	— — — — —
5	· · · · ·	0	— — — — —

这些数字的码字看起来还有些规律（相对于字母对应的码字而言）。大多数标点符号的码字采用5位、6位或7位的码长：

.	·	—	—
0	· · · · ·	[· · · · ·
1	· · · · ·]	· · · · ·
2	· · · · ·	—	· · · · ·
3	· · · · ·	+	· · · · ·
4	· · · · ·	—	· · · · ·
5	· · · · ·	—	· · · · ·
6	· · · · ·	—	· · · · ·
7	· · · · ·	—	· · · · ·
8	· · · · ·	—	· · · · ·
9	· · · · ·	—	· · · · ·

对欧洲一些语言中的重音字母以及一些有特殊用途的缩写定义了特别的码字， **SOS**就是 这样一个缩写：发送时每个字母的码字之间仅有一点的时间间隔。

如果有特制的用于发送摩尔斯电码的手电筒，你和朋友之间的交流就方便多了。这种手 电筒除了常有的开关，还有一个按钮，按压按钮就可以控制电筒的亮灭。经过练习后，你们 每分钟可以发送和接收 5~10个单词。虽然仍比交谈慢（大概每分钟 100个词左右）但已足够 用了。

当你和朋友最终熟记了摩尔斯电码时（这是唯一精通发送接收的方法），你也可以用它代 替日常用的语言。为了达到最高的速度，可以发“滴（ **dih**）”音代表点、“嗒(**dah**)”音代表划。 摩尔斯电码同样也可将文字简化为用点和划两个符号表示。

以上的关键在于“两”这个词 —“滴、嗒”两个声音，“点、划”两种方式。实际上任 何两种不同的东西经过一定的组合都可以代表任 何种类的信息。

第 2 章 编码与组合

摩尔斯电码由萨缪尔·摩尔斯（1791—1872）发明，本书后面会在多处提到他。摩尔斯电码是随着电报机的发明而产生的，电报机我们以后也还要做详尽的说明。正如摩尔斯电码很好地说明了编码的本质一样，电报机也提供了理解计算机硬件的良好途径。

大多数人认为摩尔斯电码的发送易于接收，即使你没有记住摩尔斯电码，也可以方便地借助下面这张按字母顺序排列的表发送：

A	· —	J	· — — — — —	S	— · · ·
B	— · — · · ·	K	— — — · — — —	T	— — —
C	— — — · — — — ·	L	· — — — — ·	U	— · — — —
D	— — — · · ·	M	— — — — —	V	— · — · — — —
E	·	N	— — — ·	W	— · — — — —
F	· · — — — — ·	O	— — — — — —	X	— — — · — — — —
G	— — — — — ·	P	· — — — — — ·	Y	— — — · — — — — —
H	· · — · — —	Q	— — — — — · — — —	Z	— — — — — · ·
I	· · ·	R	· — — — — ·		

接收摩尔斯电码并将其翻译回单词比发送费时费力多了，因为译码者必须反向地将已编码的“滴-嗒”序列与字母对应。例如，在确定接收到的字母是“Y”之前，必须按字母逐个地对照编码表。

问题是我们仅有一张提供“字母→摩尔斯电码”的编码表，而没有一张可供逆向查找的“摩尔斯电码→字母”译码表。在学习摩尔斯电码的初级阶段，这张译码表肯定会提供很大的便利。然而，如何构造译码表却毫无头绪，因为我们似乎无法找出这些按字母顺序排列的“滴-嗒”序列的规律。

•	E
—	T

















那么忘记那些字母序列吧，也许按照码字中“滴”“嗒”的个数来排列会是个更好的尝试。例如，仅含一个“滴”或“嗒”的摩尔斯电码序列只可能代表 E或T这两个字母之一：

• • •	I	— — —	N
• — — —	A	— — — —	M

• • • • •	S	— — — — •	D
• • • — — —	U	— — — • — — —	K
• — — — — •	R	— — — — — •	C
• — — — — —	W	— — — — — — —	O

两个“滴”或“嗒”的组合则代表了 4个字母I、A、N、M： 三个“滴”或“嗒”的序列代表了 8个字母：

最后（如果不考虑数字和标点符号的摩尔斯电码），四个“滴”或“嗒”的序列则共代表了16个字母：

	H		B
	V		X
	F		C
	U		Y
	L		Z
	K		Q
	P		O
	J		S

四张表共包括2 + 4 + 8 + 16 = 30个编码，可与30个字母相对应，比拉丁字母所需的26个字母还多了4个。出于这个原因，在最后一张表中，你可能注意到有4个编码与重音字母相对应。在翻译别人发送的摩尔斯电码时，上面4张表提供了极大的便利。当你接收到一个代表特

定字母的码字时，按其中含有的“滴”“嗒”个数，至少可以跳到其对应的那张表中去查找。每张表中，全“滴”的字母排在左上角，全“嗒”的字母排在右下角。

你注意到4张表大小的规律了吗？每张表都恰好是其前一张表的两倍大小。这其中包含的意义是：前一张表的码字后加一个“滴”或加一个“嗒”，即构成了后一张表。

可以按下面的方式总结这个有趣的规律：

2	4
3.	8
4	16

四张表中每张码字数都是前一张的两倍，那么如果第一张表含 2 个码字，第二张表则含 2×2

个码字，第三张表 $2 \times 2 \times 2$ 个码字。以下是另一种表达方式:

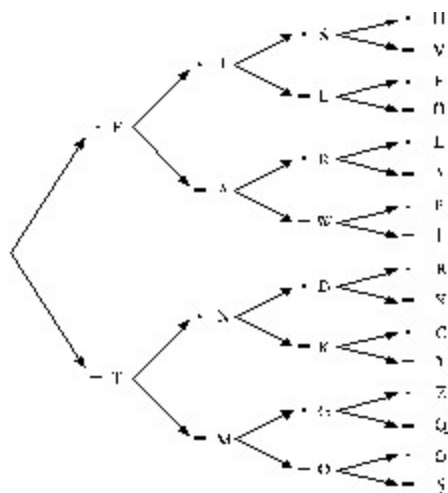
点划数	码字数
1	2
2	2×2
3	$2 \times 2 \times 2$
4	$2 \times 2 \times 2 \times 2$

当然，如果遇到数的自乘，可以用幂表示，例如 $2 \times 2 \times 2 \times 2$ 可以写成 2^4 。数字 2、4、8、16 分别是 2 的 1、2、3、4 次幂，因为可以用依次乘 2 的方法将它们计算出来。由此我们的总结 还可以写成下面的方式:

点划数	码字数
1	2^1
2	2^2
3	2^3
4	2^4

这张表简单明了，码字数是 2 的次方，次方数目与码字中含有的“滴”“嗒”数目相同。我们可以把表总结为一个简单的公式：

码字数 = $2^{\text{“滴”与“嗒”的数目}}$ 很多编码中都用到 2 的幂，在下一章中我们会看到另一个例子。为了使译码的过程更为简便，可以画出如下一张树形图：



这张表表示出了由“滴”与“嗒”的连续序列得出的字母。译码时，按箭头所指从左到右进行。例如，你想知道电码“滴-嗒-滴”代表的字母，那么从最左边开始选择点，沿箭头向右选择划，接着又是点，得出对应的字母是 R，它写在最后一个点的旁边。

如果认真考虑，会发现事先建立这样一张表是定义摩尔斯电码所必需的。首先，它保证了你不会犯给不同的字母相同码字的错

误！其次，它保证你使用了全部的可用码字，而没有使“滴”与“嗒”的序列毫无必要的冗长。

我们可以加长码字至 5 位或更长，5 位长的码字又提供了额外的 32 ($2 \times 2 \times 2 \times 2 \times 2$ 或 2^5)

个码字。一般而言，这就足够 10 个数字和 16 个标点符号使用。实际上，摩尔斯电码中的数字确实是 5 位的，但在许多其他编码方式中，5 位码字常用于重音字母而不是标点符号。

为了包含所有的标点符号，系统必须扩充至 6 位表示，提供 64 个附加编码，此时系统可表示 $2+4+8+16+32+64$ 共 126 个字符。这对摩尔斯电码而言太多了，以至于留下许多“未定义”的码字。此处“未定义”指不代表任何意义的码字，如果你接收的摩尔斯电码中有未定义的码字，就可以肯定发送方出了差错。

由于推出了下面这条公式：

码字数 = $2^{\text{“滴”与“嗒”的数目}}$ 我们就可以继续导出更长的码字数所代表的码字数目。很幸运，我们不必为确定码字数目而写出所有可能的码字，我们所要做的不过是不断地乘 2 而已：

点划数	码字数
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
4	$2^4 = 16$
5	$2^5 = 32$
6	$2^6 = 64$
7	$2^7 = 128$
8	$2^8 = 256$
9	$2^9 = 512$
10	$2^{10} = 1024$

摩尔斯电码被称为 二 元 码 （**binary code**），因为编码中仅含“滴”和“嗒”。这与一个硬 币很相似，硬币着地时只可能是正面或反面。二元事物（例如硬币）、二元编码（例如摩尔斯 电码）常常用 2 的乘方来描述。

上面所做的对二元编码的分析在数学上的一个分支—组合学或组合分析 里只能算是一个 简单的练习。传统上，由于组合分析能够用来确定事件出现的几率，例如硬币或骰子组合的 数目，所以它常用于概率统计，但它也同样有助于我们理解编码的合成与分解。

第 3 章 布莱叶盲文与二元编 码

摩尔斯不是第一个成功地将书写语言中的字母翻译成可解释代码的人，他也不是第一个 因为其编码而受到人们纪念的人，享有这个荣誉的是一个晚摩尔斯 18 年出生的早慧的法国失明少年。虽然人们对他的生平所知甚少，但就是所知的这一些却足以给后人留下深刻印象。



路易斯·布莱叶 1809年出生于法国的 Coupvray，他的家乡在巴黎以东 25英里，父亲以打造马具为生。3岁时，在这个本不该在父亲作坊里玩耍的年龄，小布莱叶意外地被尖头的工具戳中了眼睛。由于伤口发炎，感染了另一只眼，他从此双目失明。布莱叶原本注定在贫困潦倒中度过一生（正如那时大多数盲人一样），但他的聪明才智和求知欲不久即显露了出来。在本地牧师和一位学校老师的帮助下，布莱叶和其他孩子一道上了学，10岁那年又前往巴黎的皇家盲人青年学院学习。

盲人教育的一大障碍就是他们无法阅读印刷书籍。

Valentin Haüy(1745—1822)，巴黎学校的创始人，发明了一种将字母凸印以供触摸阅读的方法。但这种方法使用起来较为困难，并且只有很少的书籍用这种方法“制造”。

视力正常的 **Haüy**陷入了一种误区。对他而言，字母 **A**就是**A**，它看起来（或感觉起来）也必须像是个 **A**。（如果给他手电筒作为交流工具，他也会试图在空气中画出字母的形状，而我们已经知道这种方法并不有效。）**Haüy**也许没有意识到一种与印刷字母完全不同的编码会更适于盲人使用。

另一种可选的编码有一个出人意料的起源。法国陆军上尉 **Charles Barbier**在1819年发明了一种他自称为 *écriture nocturne*的书写体系，这种体系也被称为“夜间文字。他使用厚纸板上 有规律凸起的点划来供士兵们在夜间无声地传递口信（便条），士兵们使用尖锥状的铁笔在纸的背面刺点和划，凸起的点可以用手指感觉阅读。

Barbier体系的问题是其过于复杂。**Barbier**没有用凸起的点来代表字母表中的字母，而是用其代表声音。这样的系统中一个单词通常需要许多码字表达。这种方法在野外传递短小消息还算有效，但对长一些的文章而言则有明显不足，更不要说是整本的书籍了。

布莱叶在 12岁时就熟悉 **Barbier**方法了，他喜欢使用这些凸点，不仅因为它们易于用手指阅读，更因为它们易于书写。教室里拿着铁笔和纸板的学生可以记笔记供课后阅读。布莱叶 勤奋地工作试图改进这种编码系统。不出 3年（在他 15岁时），他创建了自己的系统，其原理直到今天还在使用。布莱叶系统有很长时间仅局限在他所在的学校使用，后来它逐渐扩散到世界各地。1835年，布莱叶染上了结核病。1852年，在他 43岁生日过后不久，他便去世了。

时至今日，布莱叶系统的改进版本甚至可以与有声录音带竞争，它为盲人提供了与书写世界联系的途径。布莱叶方法仍是适于既聋又盲的人阅读的唯一方法。近年来，随着电梯和

自动语言机的普及，布莱叶系统更加广为人知。本章将剖析布莱叶编码的编码方法及其工作原理，不过不必真正学习布莱叶编码或记住

任何东西，我们只要大概了解一下编码的本质就行了。布莱叶编码中，普通书写语言的每个字符——具体而言如数字、字母和标点符号——都被

编码成局限在 2×3 小格中一个或多个凸起的点。这些小格一般被标记为 1~6:

1	○	○	4
2	○	○	5
3	○	○	6

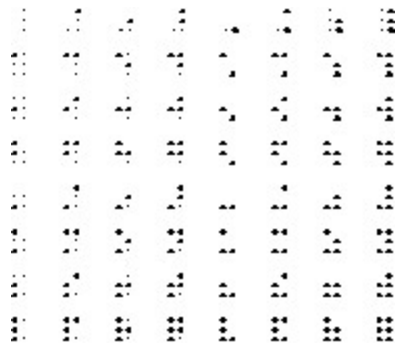
在当今实际使用中，特殊的打字机或刻印机可以在纸上打出布莱叶编码中的小点。由于在书中夹印几页布莱叶编码极其昂贵，我们使用了在通常印刷品中常用的布莱叶码

的表示方法。在这种表示方法中，小格中的 6 个点全部印刷出来，大点代表小格中的凸起点，小点则代表平滑的点。例如下图中的布莱叶字母中，点 1、3、5 是凸起的，点 2、4、6 则没有：

⠠

在这里吸引我们的问题是：点是二元的。一个特定的点不是凸起的的就是平滑的，那么 6 个点的组合数目就是 $2 \times 2 \times 2 \times 2 \times 2 \times 2$ ，或 $64(2^6)$ 。

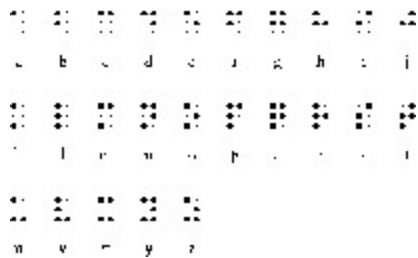
因此，布莱叶编码系统可以代表 64 个不同的码字。以下就是所有的 64 个码字：



如果我们发现布莱叶编码只用了 64个码字中的一部分，我们会疑问为什么 64个码字中有

一些不被使用；如果发现布莱叶编码使用了多于 64个的码字，则又会让人怀疑我们是否神志清醒或数字计算的真实性，2乘2是等于4吗？

分析布莱叶编码，还是从基本的小写字母开始：



	● ● ●	● ● ● ● ● ●	● ● ● ● ● ●	● ● ● ● ● ●
	● ● ● ● ● ●	● ● ● ● ● ●	● ● ● ● ● ●	● ● ● ● ● ●
	● ● ● ● ● ●	● ● ● ● ● ●	● ● ● ● ● ●	● ● ● ● ● ●

这就是布莱叶发明的布莱叶编码的基础，布莱叶还为法文中出现的重音字母设计了码字。注意，**w**没有对应的码字，这时由于在古法语中没有**w**（不必担心，这个字母最终还是会露面的）。这样算来，我们仅使用了64个码字中的25个。

通过仔细的检查，会发现上面的布莱叶编码存在特定的规律。第1行（从字母a~j）只用了小格的上面4个点——点1、2、4、5；第2行除了点3凸起外其余都与第1行相同，第3行则除了点3、6凸起外其余都与第1行相同。

在布莱叶之后，布莱叶编码在许多方面有了扩展，现在大多数英语出版物所使用的系统是二级布莱叶码。二级布莱叶码采用了许多缩写来简化编码树以提高阅读速度。以下的三行

(包括“完整的”第 3 行) 显示了下面这些词的码字:

face	am	an	on	very	men	in	have	non	is
knowledge	like	time	not	please	please	quite	some	so	that
is	with	at	you	as	and	on	of	the	with

因此，在二级布莱叶码中，短语“you and me”被写成如下形式：

到现在为止，已描述了 31 个码字——词间没有凸起点的空格和三行每行 10 个用于字母和单词的码字。这离理论上可用的 64 个码字还相距甚远。不过我们将要看到，在二级布莱叶码中，没有任何浪费的码字。

首先，我们使用 a~j 的编码加上凸起的 6 号点。它们代表词中的缩写，这其中包括 W 和另一个词的缩写：



举例来说，“about”可以用二级布莱叶码写成如下形式：

其次，可以把代表字母 a~j 的码字中的点下移一行，即仅使用点 2、3、5 和 6。这些码字根

⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋮

第一个（点 4 凸起）是重音字母标识符，其余的作为一些缩写的前缀，也用于其他用途：

点 4、6 凸起时（本行的第 5 个码字），该码字代表数字中的小数点或强调标识符，这由上下文决定。点 5、6 凸起时，码字则是与数字标识对应的字母标识。

最后（也许你正在疑惑布莱叶编码如何表示大写字母），我们用 6 号点来作为大写标识，它表明其后跟随的字母是大写的。例如，可用如下的码字写出该编码创始人的名字：

这包含大写字母标识、字母 l、缩写 ou、字母 i 和 s，空格，另一个大写字母标识，字母 b、r、a、i、l 和 e（在实际应用中，该名字还可以再删掉最后两个不发音的字母）。

总结一下，我们已经看到了 6 个元素（凸点）如何恰好形成 64 个码字。这 64 个码字根据上下文大多有双重含义，其中有数字标识以及取消数字标识作用的字母标识。这些标识改变了跟随其后的码字的含义——从字母变数字或从数字变字母。起这种作用的码字常被称为“先行码/前置码”或“转义码”，它们更改其后字符的含义直至更改作用被取消。

大写标识表示其后的字母（也仅有字母）应写成大写，这种码字被称为“换码代码”。“换码代码”使你“避免”那种单调的、常规的码字解释，而转入一种新的解释方法。在以后几章中可以看到，当把书面语言转换为二数码字时，“换码代码”和“转义码”的使用是很普遍的。

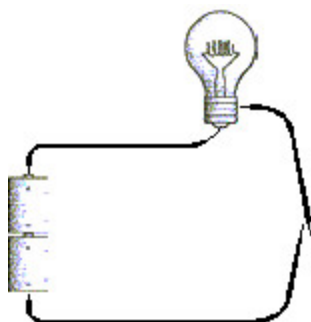
第 4 章 手电筒剖析

手电筒的用途极为广泛，用于在黑暗的遮盖物里阅读和用于发送编码消息只是两个用途最明显的方面。最普通的家用手电筒也能在教学演示中说明神秘物质电（**electricity**）时扮演中心角色。

电是一种令人称奇的现象，尽管它已得到普遍应用，但依然还保持着很大的神秘性，即使对那些自称已经弄清楚它的工作原理的人而言也是这样。但恐怕不管怎么样，我们都必须好好努力钻研一下电学。幸运的是，我们只需要明白一小部分基本概念就可以理解它在计算机中是怎样应用的。

手电筒当然是一种大多数家庭都拥有的较简单的电器。拆开一支有代表性的手电筒，你会发现它包括一对电池，一个灯泡，一个开关，一些金属片和一个把所有零件装在一起的塑料筒。

只用电池和灯泡，就可以自己做一个简单的手电筒。当然，还需要一些短的绝缘线（末端的绝缘皮除掉）和足够多的连接物：



注意上图右边两个松开的线端（头），那就是开关。如果电池有电并且灯泡也没有烧坏的话，接触两个线端，灯就亮了。

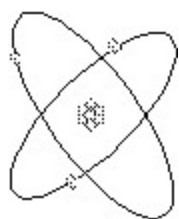
这是我们要分析的头一个简单电路，首先要注意的是电路是一个回路。只有从电池到电线、到灯泡、到开关、再回到电池的路径是连续畅通的，灯泡才会亮。电路中任何一点断开都会引起灯泡的熄灭。开关的目的就是控制电路开闭这个过程。

电路环接的特性提示我们有某种物质在电路中循环移动，可能与水在水管里流动有某些相似。“水与水管”的类比常用来解释电的工作机理，但最终它也像其他类比一样不可避免地解释不下去了。电在宇宙中是独一无二的，必须用它的术语来解释它。

在对电的工作的理解中，最流行的科学理论是电子理论（**electron theory**），该理论认为电起源于电子的运动。

众所周知，一切物质——我们能看到、感觉到的东西——（通常）是由极其微小的被称为

原子的东西构成。每一个原子是由三种微粒构成的，即中子、质子和电子。你可以把原子画成一个小的太阳系，中子和质子固定在原子核内而电子像行星环绕太阳一样围绕原子核运动：



需要解释一下的是该模型与你在一个放大倍数足够大的显微镜下看到的真正原子不是一模一样的，它只是一个示例模型。

图中原子包含 3 个电子、3 个质子和 4 个中子，说明这是一个锂原子。锂是已知的 112 种元素之一，它们的原子序数由 1~112。一种元素的原子序数是指元素的原子核中质子的个数，通常也是其电子数。锂的原子序数为 3。

原子能够通过化学合形成分子，分子与组成它的原子的性质通常是不同的。比如水分 子包含两个氢原子和一个氧原子（即 H_2O ）。显然水既不同于氢气，也不同于氧气。同样，食 盐分子由一个钠原子和一个氯原子构成，而钠和氯都不可能成为法国馅饼的调味品。

2

氢、氧、钠、氯都属于元素，水和食盐都属于化合物。但是盐水是一种混合物，而不是 化合物，因为其中水和食盐都保持它们各自的性质不变。

一个原子的电子数通常等于其质子数。但在某种特定环境下，电子能从原子中电离出来，这样电就产生了。

单词electron和electricity都源于古希腊词 $\epsilon\lambda\epsilon\kappa\tau\rho\omicron\nu$ (elektron),你可能猜它的意思就是“极其微小而不可见的东西”。但事实并非如此—— $\epsilon\lambda\epsilon\kappa\tau\rho\omicron\nu$ 的真正意思是“琥珀”，一种玻璃状的硬质树液。这个看似不相关的词源来自于古希腊人所做的实验，他们用琥珀与木头相摩擦而产生我们今天所说的静电。在琥珀上摩擦木头使木头从琥珀获得电子，结果木头所含的电子数多于质子数而琥珀所含的电子数小于质子数。在更多的现代实验中，地毯能从鞋底获得电子。

质子和电子具有带电荷的特性，质子带正电荷（+）、电子带负电荷（-）。中子是中性的，不带电。即便我们用加减号来标明质子和电子，但符号并不表示算术运算中的加号和减号的意思，也不表示质子拥有某些电子所不具备的东西。使用这些符号仅仅表示质子和电子在某个方面性质相反。这个相反的特性也正表明了质子和电子是如何相互关联的。

当质子数与电子数相等时，它们是最适合和最稳定的。质子数与电子数的不平衡会导致它们趋于平衡。静电火花就是电子运动的结果，是电子从地毯通过你的身体再流回到鞋子的过程引起的。

描述质子和电子关系的另一条途径是注意观察异电性相吸同电性相斥的现象，但光凭看原子结构图我们是不能猜想到的。表面上看原子核中挤在一起的质子是互相吸引的。质子是通过比同性斥力大的某种力聚合在一起的，这种力叫强内力。释放核能的原子核裂变就是由于强内力导致的。本章只讨论通过得失电子获得电（电能）的问题。

静电不只存在于手指触摸门把手时闪出的火花之中。暴风雨时，云层的下层积累电子而

云层的顶层失去电子，闪电的瞬间，电子的不平衡马上消失。闪电正是大量的电子迅速从一端转移到另一端的结果。

手电筒电路中的电能显然比电火花或闪电之中的电能要好利用得多。灯泡能稳定持续地亮是因为电子并不是从一点跳到另一点。当电路中的一个原子把一个电子传给邻接的另一个原子时，它又从另一个邻接的原子获得电子，而这个原子又从它的一个邻接原子获得电子，如此依次循环。可见电路中的电就是从原子到原子的电子通路。

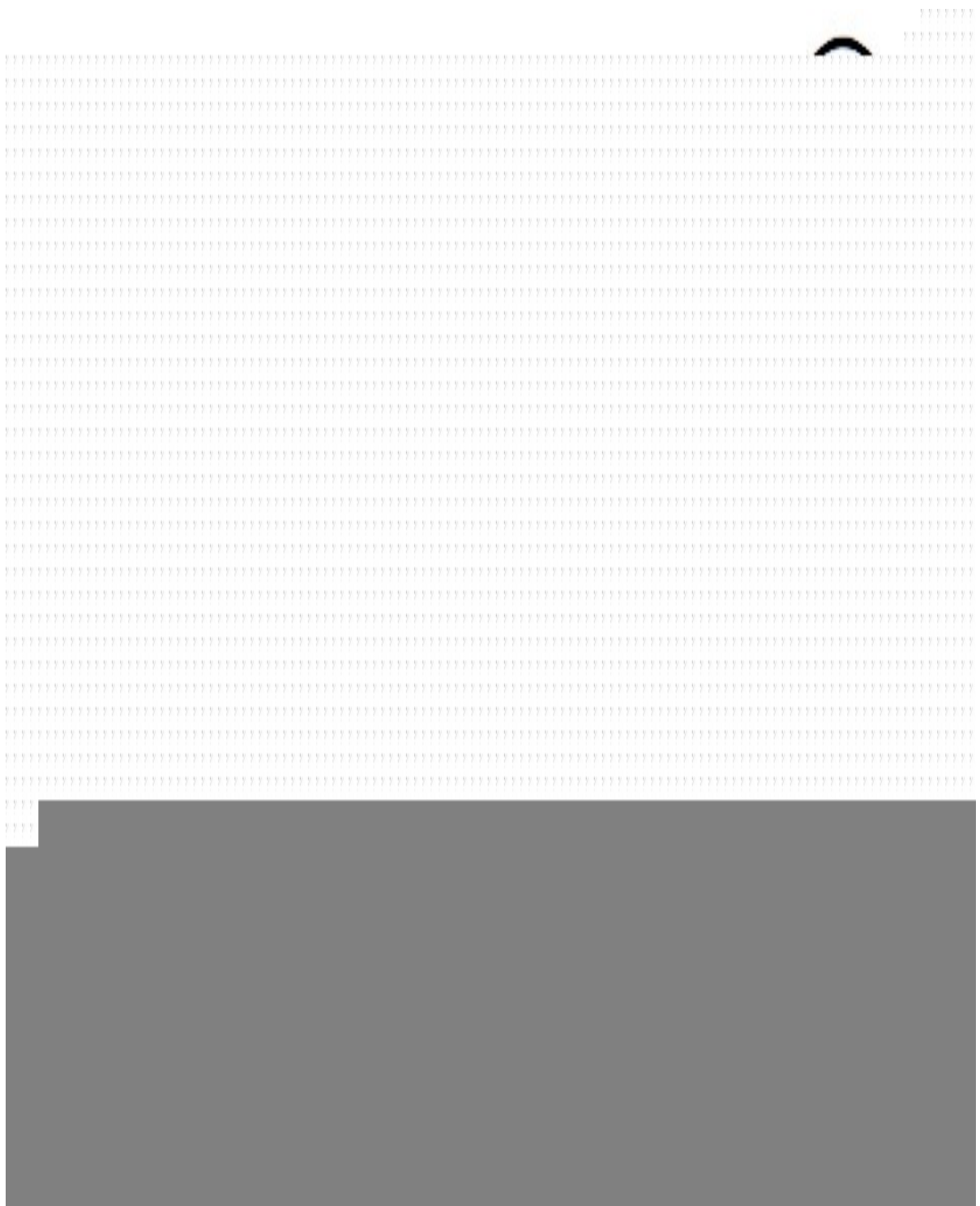
这不可能自发形成。仅仅只把一些破旧的电路材料连接在一起是不可能有电能产生的，需要某种可以激发电子环绕电路移动的物质。再分析一下前面所画的简单手电筒电路图，可以肯定激发电子运动的既不是电线，也不是灯泡，那么最有可能的就是电池了。

几乎每一个人都多少了解手电筒里所用电池的类型方面的一些知识：

- 它们都呈管状，且大小不同。比如有 D、C、A、AA 和 AAA 等型号。
- 无论电池大小怎样，它们都被标有“1.5 伏”。
- 电池的一端是平的，标有一个负号（-）；另一端中间有一个小突起，标有一个正号（+）。
- 要想设备正常工作，就要正确安装电池（注意电池极性）。
- 电池的电能最终将用尽。有的电池可以充电，有的不行。
- 由此可以猜测，电池是用某种奇特的方式产生电能。所有的电池中都发生着化学反应，一些分子裂变成其他分子或者结合形成新的分子。电

池中有化学物质，这些化学物质就是用来起反应，从而在标有（-）的电池的一端（称为负极或阴极）产生多余的电子而在电池的另一端（称为正极或阳极）需要得到电子。这样，化学能转化为电能。

只有当某种特别的电子通过某条途径从电池负极出发，然后再传送到正极时，化学反应才能发生。因此假如一节空电池放在那里，那么什么事也不会发生（事实上，化学反应还是在进行的，只是速度极慢）。只有一条电路能将电子运离负极又为正极提供电子时，反应才会发生。电子在下图电路中是沿逆时针方向运动的：

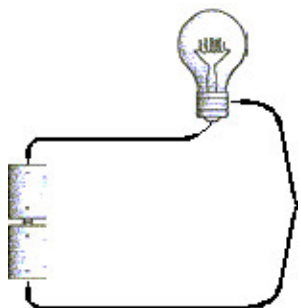


如果不是基于这个简单的事实：所有的电子，不管来自什么地方，都是一模一样的，否则，来自电池的化学物质里的电子就不可能如此随意地与铜导线的电子混合在一起的。铜导

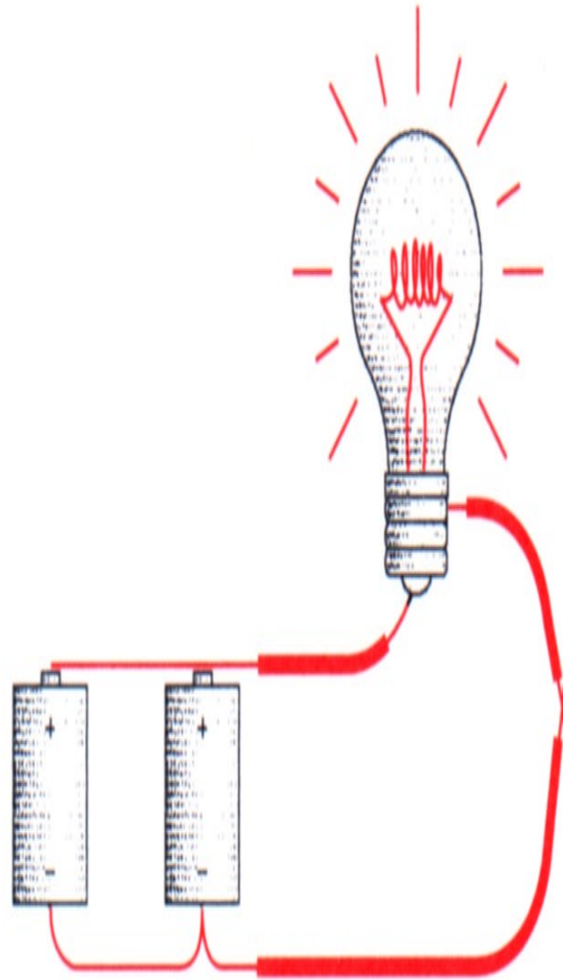
线的电子与任何其他电子是没有区别的。注意，两个电池都是向着同一个方向。放在下面的电池的正极从上面电池的负极获得电

子，这样两个电池就好像结合形成了一个更大的电池，这个大电池一端为正极，另一端为负极，其电压是 3 伏而不是 1.5 伏了。

如果把电池中的一个倒置，电路就会连不通，如下图所示：



在化学反应中，两个电池的正极都需要获得电子，但由于它们相互接触，电子无法通过某种途径到达它们。如果两个电池的正极连上了，那么它们的负极也应该连上，如下图所示：



这样的电路还是能连通。电池的这种连接方法称为并联，前一种连接方法称为串联。并联后的电压与单个电池电压同样都是 1.5 伏。并联后的灯仍然可能亮，但不如串联时亮度大，不过电池的寿命将会是串联时的两倍。

通常认为电池为电路提供电能，但同样也可以认为电路为电池化学反应的发生创造了条件。电路将电子从负极传送到正极。电路中的化学反应将一直进行到所有的化学物质耗尽，这时你就需要换电池或是给电池充电了。

电子从电池的负极到正极流过了导线和灯泡。为什么需要导线？电不能通过空气传导吗？噢，可以说能，也可以说不能。电能够通过空气导通（尤其是潮湿的空气），否则也观察不到闪电。但电不能很轻易地流经空气。

一些物质的导电能力比其他物质的导电能力明显要好。元素的导电能力取决于它内部的

原子结构。电子绕核旋转是在不同的轨道上的，这些轨道称为层。最外层只有一个电子的原子最容易失去那个电子，这正是导电需要具备的性质。这些物质易导电因而被称为导体。铜、

银和金都是良好导体，这三种元素位于元素周期表的同一列不是巧合。铜是最常用的导线材料。

导电物质的对立物质称为绝缘物质。一些物质阻碍电的能力比其他物质阻碍电的能力强，这种阻碍电的能力称为电阻。如果一个物质有很大的电阻——说明它根本不能导电——它就被称为绝缘体。橡胶和塑料都是很好的绝缘体，因而它们常用来做电线的绝缘皮。在干燥空气的情况下，布料和木材也是很好的绝缘体。其实只要电压足够高，任何物质都能导电。

铜的电阻很小，但它仍有电阻。导线越长，电阻越大。如果你用数里长的导线连接手电

筒，导线的电阻将会大得令手电筒不亮。导线越粗，电阻越小，这可能有点违反直觉。你也许认为粗的导线需要更多的电来“充满它”。而事实上，导线越粗，电子越容易通过它。我已经提到过电压，只是还没有给出它的定义。一节电池为 1.5 伏特意味着什么呢？实际上，电压

——得名于 Count Alessandro Volta(1745—1827)，他于 1800 年发明了第一节电池——是初等电学中较难理解的概念之一。电压表征电势能的大小，无论一节电池是否被连通，电压总是存

在的。

假设有一块砖头。如果把它放在地上，它的势能很小。当你把它举起至离地面 4 英尺高时，它的势能就增加了。你只要把砖块扔下，就能感觉到势能的存在。当你在一座高楼的顶层举着砖块时，它的势能更大。上面三个例子里，你只是拿着砖块而什么也没做，但砖块的势能却不同。

电学里更早的一个概念是电流。电流取决于电路中飞速流动的电子的数量。电流用安培来度量，它得名于 André Marie

Ampère(1775 —1836), 一般简称安, 比如“ 10安的保险丝”。当6 240 000 000 000 000 000个电子在 1秒内流过一个特定的点时, 就是 1安培电流。

用水和水管作个类比。电流与流经水管的水量很相似, 而电压类似于水压, 电阻类似于

水管的粗细程度—水管越小、阻力越大。因此水压越高, 流过水管的水量越大; 水管越小, 流过它的水量就越少。流过水管的水量(电流)与水压(电压)成正比而与水管的阻力(电阻)成反比。

在电学中, 如果知道电压和电阻的大小, 就可计算出电流的大小。电阻—物质阻碍电流通过的能力 —用欧姆度量, 得名于 Georg Simon Ohm (1789—1854), 他提出了著名的欧姆定律, 定律中表述

这里 I 表示电流， E 表示电压， R 表示电阻。举个例子，让我们看一节空置的电池：

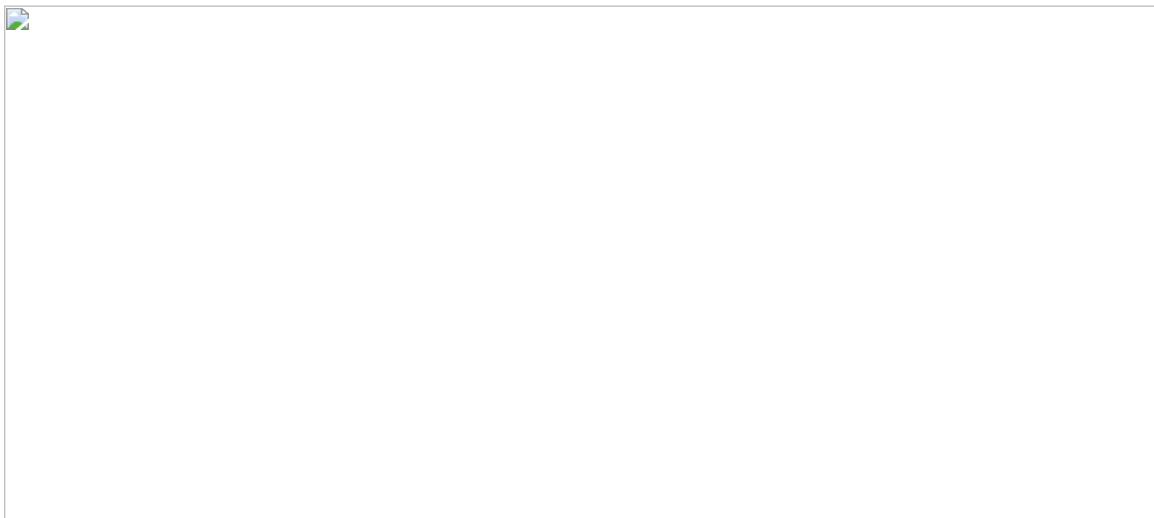
$$I=E/R$$



它的电压 E 为 1.5 伏，这是电势能。因为电的正负两极只被空气导接，因而电阻（用 R 表示）非常、非常大，这就意味着电流 I 等于 1.5 除以一个巨大的数，电流几乎为 0。

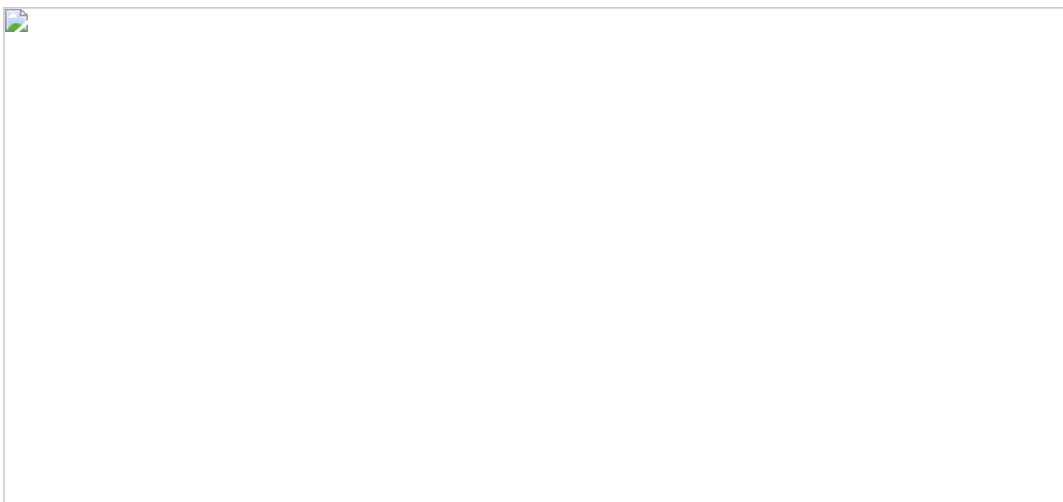
现在用一根短铜导线连接电池的正负两极（从现在开始，本书中导线外的绝缘皮不再表

示出来）：



我们已经知道这是短路。电压仍是 1.5 伏，但电阻很小，这时电流等于 1.5 除以一个很小的数，

也即意味着电流很大。很多很多的电子将流过导线。实际上，电流将受到电池物理大小的限制。电池不可能导通如此大的电流，且实际电压也将低于 1.5 伏。如果电池足够大，导线将会发热，因为电能转化为了热能。如果导线变得很热，它将会发光（辉光放电）甚至可能熔化。



绝大部分电路都介于这两个极端之间。可以把它们统一表述为如下图：

电气（子）工程师用折线来表征电阻。这里它表示电阻不是特别大，也不是特别小。如果导线的电阻很小，导线将发热发光，这就是白炽灯的工作原理。白炽灯泡是由美国

最著名的发明家托马斯·爱迪生（1847—1931）发明的。在他致力于发明灯泡的时候（1879

年），这个思想已被普遍接受并且同时还有不少其他发明家在研究这个问题。灯泡里的细线叫灯丝，通常用金属钨做成。灯丝的一端连在基座底部的尖端，另一端连

在金属基底的一个侧面，用一个绝缘体将它与尖端分开。细线的电阻使它发热。如果暴露在空气中，钨就会由于达到燃烧温度而烧起来。但在灯泡的真空中，钨丝就发亮了。

大多数普通手电筒用两节电池组成一组，总电压是 3.0 伏。且选用电阻大约为 4 欧姆的灯泡。这样，电流等于 3 除以 4 即 0.75 安培，也就是 750 毫安。这就意味着每秒钟有 4 680 000 000 000

000 000 个电子通过灯泡。（注意，如果你用欧姆表直接测量手电筒灯泡的电阻，你只会得到一个比 4 欧姆小得多的结果。这是因为钨的电阻还与它的温度有关系，温度越高，电阻越大。）你可能已经发现，你买回家的灯泡上标记了特定的瓦特数。瓦特这个名词取自于著名的

蒸气机发明家詹姆斯·瓦特（1736—1819）。瓦特是功率 P 的单位，它用下式计算

$$P=E \times I$$

手电筒是 3 伏，0.75 安培，那么灯泡的功率就要求 2.25 瓦特。

家用照明灯大约为 100 瓦特，这是为家用电压 120 伏设计的。在这种情况下，电流为 100 瓦除以 120 伏即大约 0.83 安培。因此，100 瓦特灯泡的电阻为 120 伏除以 0.83 安培即 144 欧姆。

到此，我们大致分析了手电筒的每一个组成部分——电池、导线和灯泡。但是我们遗漏了一个最重要的部分、对，是它的开关。开关控制电路的开闭。当开关允许电流流动时，我们说它是开的或合上的，而关的或断开的开关是不允许电流流动的。（这里所表示的开、关的状态正好与门相反，合上的门不允许事物通过的，而合上的开关允许电通过。）开关或开或关，电流或有或无，灯泡或亮或不亮，就像摩尔斯和布莱叶发明的二进制一样，简单的手电筒或亮或不亮，它没有中间状态。二进制与电气电路之间的相似性将在后面的章节中起很大作用。

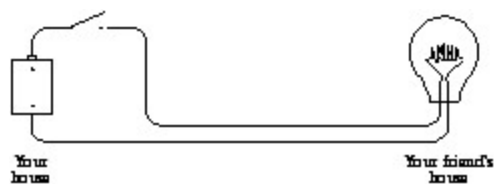
第 5 章 绕过拐弯的通信

你12岁了。一天，你最要好的朋友一家要搬到另一个镇上去了。此后，你经常和他在电话里聊天，但电话交谈与那些后半夜的手电筒摩尔斯电码会话完全不一回事。住在你隔壁的另一个好朋友最终成为你新的最要好的朋友。现在到了该教你的新朋友一些摩尔斯电码，让后半夜的手电筒重新亮起来的时候了。

问题是你的新朋友的卧室窗户与你的不是面对面的。房子是挨着的，卧室的窗户都朝着同一个方向。除非你想办法在室外支起一些镜子，否则手电筒现在是不能适用来在黑夜中通信的。

怎么办呢？现在，你可能已经知道有关电的一些知识了，因此你决定用电池、灯泡、开关和导线来

做自己的手电筒。最初的实验中，你在你的卧室里接好电池和开关。两条导线接出你的窗子，跨过篱笆，再接进你朋友的卧室，并在那里再连好灯泡：



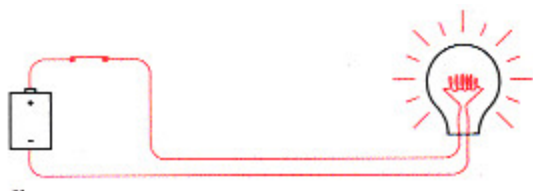
你的房子 你朋友的房子

尽管图中只示意了一节电池，但实际上你可能得用两个。在下面和以后的图中，用下图 表示断开的开关：



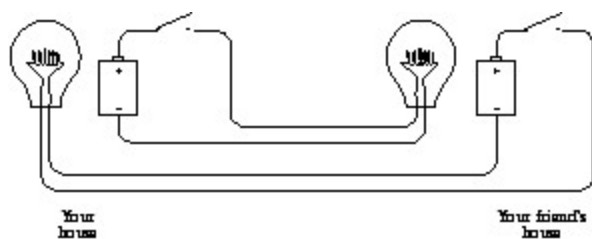
用下图表示闭合的开关：本章的手电筒与上一章中手电筒的工作原理是相同的，尽管本章的手电筒中连接组件的

导线要长得多。当你闭合开关时，你朋友那边的灯泡就亮了：



你的房子 你朋友的房子

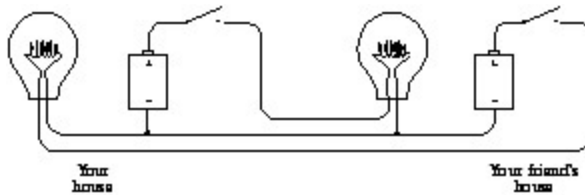
现在你可以用摩尔斯电码来发送消息了。一旦有一个手电筒起作用，你可以做另一个远距离手电筒，好让你的朋友可以发送消息给你：



你的房子 你朋友的房子

祝贺你! 你已经装上了一个双向电报系统。你可能注意到这两个相似的电路彼此完全独立 而没有联系。理论上, 你可以给你的朋友发送消息而同时你的朋友也可以给你发送消息 (尽管对于你的大脑而言, 同时阅读和发送消息可能比较困难)。

聪明的你发现如下改装电路能让你节省 25%的导线:

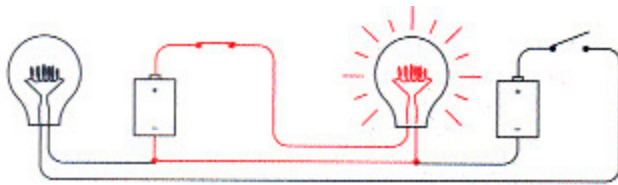


你的房子 你朋友的房子

注意，现在两个电池的负极接在一起了。两个回路（电池到开关到灯泡再到电池）仍是独立工作，尽管它们连在一起像连体双胞胎。

这种连接叫公用连接。在这个电路中，公用部分从左端灯泡和电池的接合点直到右端灯泡和电池的接合点。图中接合点用黑点标记出来了。

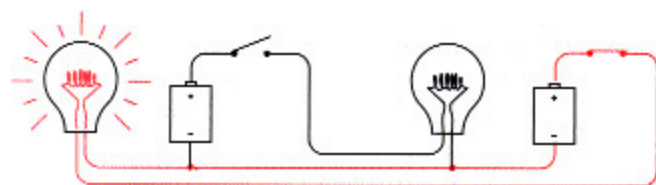
进一步分析一下。首先当你按下开关，你朋友那边的灯就亮了。图中浅色回路中有电流流过：



你的房子 你朋友的房子

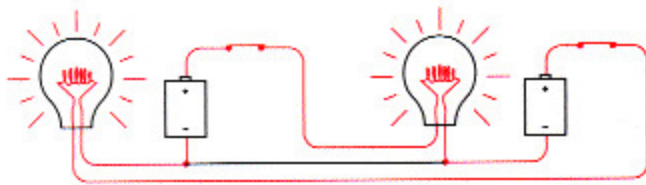
在电路的其余部分里没有电流流过，因为没有了可让电子通过的回路。当你不发消息而你的朋友发消息时，你朋友房间里的开关控制你房间里灯泡的亮灭。在

下图浅色回路中有电流流过：



你的房子 你朋友的房子

当你和你的朋友想要同时发消息时，有时两个开关同时断开，有时一个断开一个闭合，有时两个同时闭合。在最后一种情况下，电路中电的流动如下图所示：



你的房子 你朋友的房子

公用部分 (两个接合点之间) 没有电流流过。通过公用部分把两个独立电路连接成一个电路，已经把两栋房子之间的四条导线减少到

了三条，也即减少了 25% 的导线开支。如果不得不接很长距离的线路，我们可能会想到再减少一根导线。但不幸的是对于 1.5 伏

的D号电池和小灯泡，这是不合适的。如果用的是 100伏的电池和大得多的灯泡时，那就有办法了。

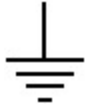
这是个窍门：如果你要搭建电路的公用部分，你不需要任何导线。你可以用另外某种东西取代它。你所用的取代物是一个直径大约为 7900英里，由金属、岩石、有机物等多为无生命的物质组成的巨大球体。它就是地球。

上一章描述的良好导体中有银、铜和金。事实上，地球不是一个很好的导体，尽管某些部分（如沼泽）的导电性能比其他部分（如干沙漠）要好得多。但我们知道导体越大越好，一根很粗的导线比一根很细的导线要强得多。这是地球的优势，它的确非常非常大。

要用地球做导体，并不是把一根小细线插到马铃薯旁边的地里就可以了。你还必须使用某种东西以维持和地球的真正接触，这也就是需要一个大面积的导体。一个很好的解决办法是用一根至少 8英尺长， 1/2英寸粗的粗铜柱，它能提供与地面 150平方英寸的接触。你可以用一个锤子把它砸进地下，然后再接一根导线。如果你家的水管是铜质的，且从房子外的地下接进来的话，那么你只要把一根导线与水管相连就可以了。

与地球的电性连接（也就是我们常说的接地）在英国叫 **earth**，在美国叫 **ground**。用 **ground** 可能会引起一点点儿误会，因为它也经常用来指电路的公用部分。本章除非特别声明，否则 **ground** 都指与地球的物理连接。

画电路图时常用下面这个符号表示接地：



电气工程师们使用这个符号是由于他们不喜欢费时间画一个埋在地下的 8 英尺长的铜柱。 让我们来看看它是怎么工作的。从分析单回路开始：



你的房子 你朋友的房子

如果你使用的是高压电池和大灯泡，你只需要在你和你朋友的房子之间接一根导线，因为你可以用大地来做导体：



你的房子 你朋友的房子

当你断开开关，电子的流动如下图所示：



电子从你朋友房子的地下出发，通过灯泡、导线和你房间里的开关，然后进入电池的正极。电子由电池的负极进入地下的。

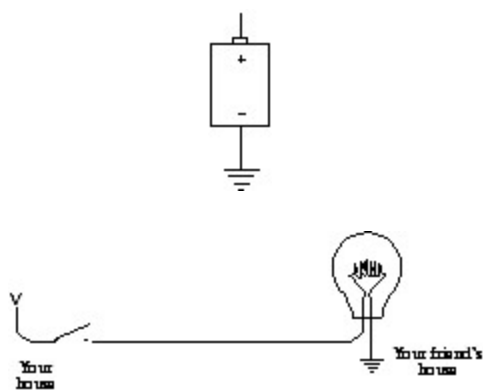
也许你还真的很想看到电子从埋在你家后院的 8 英尺长的铜柱进入地下，飞速地通过大地 到达埋在你朋友家后院的铜柱。

但是当你考虑到地球在为世界上数以千计的电路完成此功能时，你也许会问：这些电子 怎么知道该到哪儿去呢？显然它们不知道。这里要用地球的一个特殊性质来解释。

是的，地球是一个巨大的导体，但它同时也是电子的来源和仓库。地球对于电子而言就 好像大海对于水滴而言。地球的确是电子无尽的源头，也是电子巨大的存储池。

但是地球也有电阻，这就是为什么如果用 1.5 伏的 D 号电池和手电筒灯泡就不能用接地来减少电路开支的原因。地球对于低电压电池而言电阻实在太大了。

你可能注意到上面两张画了电池的图中，电池的负极接地了：



以后将不再画接地的电池，而用代表电压的字母 **V** 来代替它。单回路灯泡电报机现在如下 图所示：

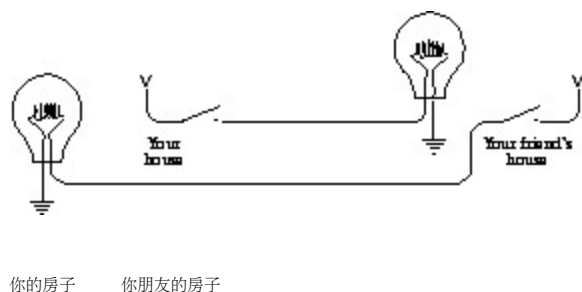
你的房子

你朋友的房子

V代表电压，但它也可以表示吸取器。把V看成电子吸取器，把大地看成电子的海洋，电子吸取器从地下吸取电子，放入回路，使之工作（比如点亮灯泡）。接地有时也被看成零电势，意味着没有电压存在。电压——像早先解释的——是一种电势能，就像悬浮的砖块具有势能一样。零电势就好像摆在地上的砖块——它不能再往什么地方掉下去了。

在第4章中，我们注意到的一件首要的事情是电路是一个回路。新电路看起来一点儿都不像回路，但它仍然是回路。你可以用负极接地的电池代替V，然后用一根线把所有有接地符号的地方连起来，你将得到与本章开始时一样的电路图。

因此，通过一对铜柱（或是自来水管）的帮助，可以只用两根跨越你和你朋友房子之间篱笆的导线就建立起了双向摩尔斯电码系统：



这个电路与先前的三线配置电路功能相同。本章已经迈出了通信改革中的关键性一步。最初，我们只能通过直线视觉和在手电筒的

可见范围内进行摩尔斯电码通信。使用电线，不仅突破了直线视觉的限制，而且通过建立系统来绕过拐弯进行通信，我们还

摆脱了距离的限制。只要搭造更长更长的线路，就可以越过成百上千英里进行通信。对了，这还不太准确。尽管铜是电学上很好的导体，但它不是最完美的。导线越长，电

阻越大；电阻越大，电流越小；电流越小，灯泡越暗。那么导线可以造多长呢？因情况而定。假设你正在使用原来四根线的双向电路，无接地

和公用，并且还用手电筒和灯泡。为了节省开支，你先从电器行买了一些 20号规格的电话线，每100英尺\$ 9.99。电话线是用来连接你的扩音器和立体声系统的。它有两根导线，因此它是电报系统的上佳选择。如果你的卧室与你朋友的卧室不到 50英尺远，只用一捆电话线就够了。

美国的导线粗细规格为 AWG。AWG数越小，导线越粗，电阻越小。你所买的 20号规格电话线直径大约 0.032英寸，每 1000英尺大约 10欧姆电阻，这样对于卧室之间 100英尺长的回路电阻为1欧姆。

这并不坏，但如果要连上英里的线呢？线的总电阻将达到 100欧姆以上。回想一下上一章

中，灯泡电阻仅为 4欧姆。利用欧姆定律，可以很容易地计算出电路中的电流不再是以前的 0.75安（3伏除以4欧），而是比 0.03安还小（3伏除以100欧以上）。几乎可以肯定，电流的大小不够点亮灯泡。

使用粗线是一个很好的解决方法，但价格太昂贵。10号规格线（电器行的汽车电路耦合线价格为每 35英尺\$ 11.99，而且你需要双倍长度因为它只有单线）大约 0.1英寸粗，1000英尺为1欧姆，即 1英里5欧姆。

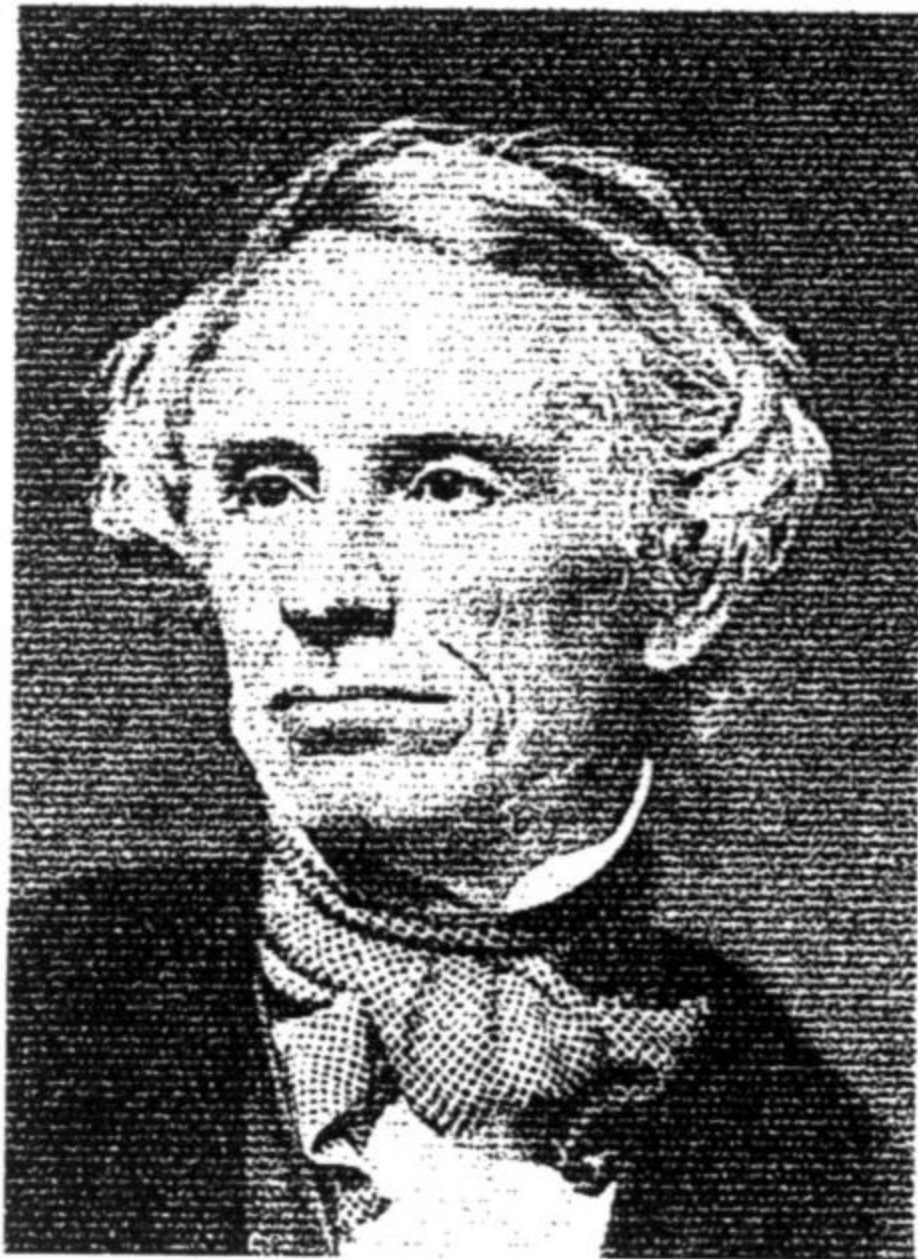
另一个解决办法是增加电压，使用大电阻灯泡。比如使用 120伏电压的 100瓦家用照明灯泡的电阻为 144欧姆。电线的电阻对于整个电路电流的影响将大大减小。

接下来的是 150年前，人们在美洲和欧洲之间搭建第一个电报系统时所面临的问题。不管 电线多粗，电压多高，电报线还是不能无限延长。根据计划，工作系统的极限为 200英里。这 与纽约和加利福尼亚间的上千英里距离相差太多。

这个问题的答案——不是为手电筒，而是为过去的嘀嗒电报——虽说是一个简单易行的设备，但是通过它，整个计算机得以构造。

第 6 章 发报机与断电器

1791年，萨缪尔·摩尔斯生于马萨诸塞州的查尔斯顿镇，该镇是邦克山之战的地点，也是波士顿东北重镇。摩尔斯出生那年，美国宪法刚实施两年，乔治·华盛顿出任美国第一个任期的总统职务。Catherine大帝统治俄国。路易十六世和 Marie Antoinette在两年后的法国大革命中被送上断头台。1791年，莫扎特完成了《魔笛》，他的最后一部作曲，次年于35岁时去世。



摩尔斯在耶鲁受过教育，又在伦敦学过艺术，他是位著名的肖像画家。他的作品《General Lafayette》(1825)珍藏于纽约市政大厅。1836年，他曾参与过竞选纽约市市长且获得了 5.7% 的选票。他也是早先的摄影术狂热爱好者。他从 Louis Daguerre 本人那儿学习了银版相片的制作，制造出了美国第一批用银版

照相术制成的相片， 1840年，他把这个手艺传授给了 17岁的 **Mathew Brady** 。此人以及他的同事后来为美国内战、亚伯拉罕·林肯和摩尔斯本人留下了一些很有纪念价值的照片。

这些只是一个多职业生涯者的足迹。摩尔斯最著名的贡献 在于他发明了电报和以他名字命名的编码。

世界范围内的即时通信我们已经很熟悉， 但它是当今新技

术发展的结果。 19世纪早期，你可以即时通信和远距离通信，但不能同时达到两个要求。即时通信只能限制在你的声音能达到（没有扩音器可用）或是你的眼睛能看到（也许得用望远镜）的范围；远距离通信则要花时间用信件通过马车、火车或者轮船的方式来实现。

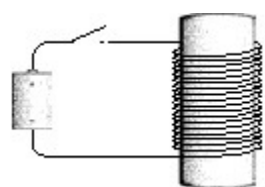
在早于摩尔斯发明的年代里，人们曾做过许多加速远距离通信的尝试。一种技术上简单

的方法是雇佣一批人接力，站在山顶上用旗语信号通信。技术上稍微复杂一点儿的方法是使 用巨大的带有可动手臂的装备，原理与旗语相同。

电报思想的正式成形是在 19世纪早期。 1832年在摩尔斯开始试验之前，已经有其他科学 家在做一些试探。原理上讲，电报思想很简单：你在线的一端做某些事引起线的另一端发生 了某些事。这正是上一章用远距离手电筒所做的事情。但摩尔斯不可能使用灯泡作为他的信

号设备，因为实用性灯泡直到 1879年才发明出来。摩尔斯使用的是电磁现象。如果你取一只铁棒，用细导线将它绕几百圈，然后让电流通过导线，铁棒变成了磁铁，

这时它就能吸引其他的铁和钢。（电磁铁上细线的电阻足够大以防止电磁铁形成短路。）移开 电流，铁棒的磁性消失：



电磁铁是电报的基础。一端上开关的闭合引起另一端上的电磁铁产生一些动作。摩尔斯最早的电报机比后来改进的要复杂得多。摩尔斯认为电报系统应该在纸上实际写

点儿什么 (这就像后来的电脑使用者描述的“生成一个硬拷贝”)。这当然不必是文字，因为文字太复杂，但某些字符应该记录下来，或曲线或点或划。注意，摩尔斯坚持要用纸记录下发报内容的这种想法，与 Valentin Haüy 要求盲人书籍应该使用突起的字母文字一样。

尽管摩尔斯早在 1836 年就告知专利局他已经成功地发明了电报，但直到 1843 年，他才说服议会为此设备的示范表演出资赞助。1844 年 5 月 24 日是有历史意义的一天，Washington 和 马里兰州巴尔的摩之间的电报线成功地传送了圣经上的一句话“ What hath God wrought! ”。

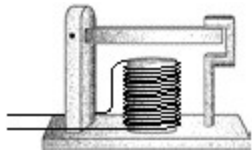
传统电报机发送消息的核心部分如下图所示：



尽管外观比较怪，但它只是一个为高速开合（闭）设计的开关，称为“按键 / 按钮”。长时间按键最舒适的方式是在手掌的拇指、食指和中指之间握住把手，然后敲击。短时间敲击形成摩尔斯电码的点，长时间敲击形成摩尔斯电码的划。

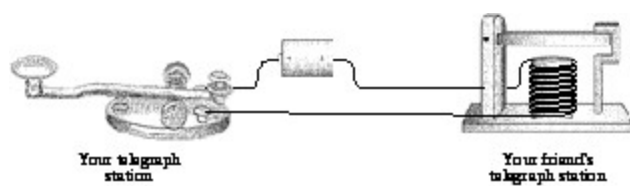
线的另一端是一个接收机，其基本结构是一个电磁铁吸拉一根金属拉杆。起初电磁铁控制的是一支笔，当由小装置控制的机械通过弯曲的弹簧缓慢地拖拉一卷纸时，相连的笔上下蹦弹将点划记录在纸上，懂得摩尔斯电码的人再将点划翻译成字母和文字。

当然，人是会偷懒的。电报机使用者很快发现只要简单地利用笔跳上跳下的声音他们就能翻译编码。笔的装置最终被撤消，代替的是传统电报机的发声装置，称为“发声器 / 音响器”，结构如下：



当电报机的键按下时，发生器的电磁铁将可动棒拖下发出“滴”的声音；当键放开时，棒弹回初始位置，发出“嗒”的声音。快速的“嘀嗒”为点，慢速的则为划。

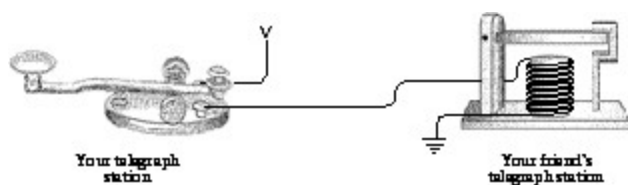
按键、发声装置，电池和一些导线可像上一章所述手电筒电报一样连接起来：



你的电报站 你朋友的电报站

我们已经知道，两个电报站之间不需要两根线。如果大地作为另一半回路的话，一根线就足够了。

如上一章所做，我们用字母 **V** 代替接地的电池，因此最终的单向设置如下图所示：



你的电报站 你朋友的电报站

双向通信只不过再需要一个按键和发生器。与上章所做相似。电报的发明真正标志着现代通信的开始。人类首次能够在眼、耳的范围之外以快于马奔

跑的速度通信。发明中使用的二代码是其精华所在，但在后来的电子和无线电通信中，包括电话、收音机和电视，二代码都没有用到，只到最近二代码才出现在计算机、CD盘、DVD盘、数字卫星电视广播和高清晰电视中。

摩尔斯的电报机战胜了其他设计，部分原因是它对不好的电线状态的容忍度比较大。假如

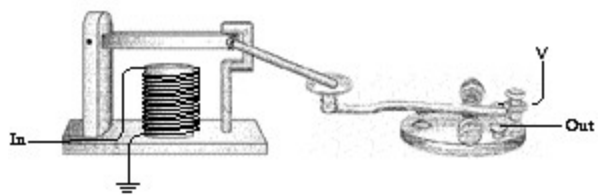
你在按键和发声装置之间接一根线，该电报机通常可以工作，但其他电报系统却不具备这样的容忍性。但正如上章所谈及的，最大的问题在于长距离导线的电阻。尽管一些电报线使用高达300伏的电压能在300英里的范围内工作，导线还是不能无限延伸。

一个明显的解决办法是使用转发（中继）系统，也称继电器系统。大约每200英里就让某位发报者通过发声装置接收消息再用按键发送出去。

现在想像一下你已被某电报公司雇佣为转发系统的工作人员。他们把你放在纽约和加利福尼亚之间某个地方的一间简陋得只有一张桌子和一把椅子的小屋里。一根导线从东边的窗户进来连到发声装置上。你的按键连在电池和从西边窗子出去的导线上。你的工作是接收来自于纽约的消息然后把它们发送到加利福尼亚。

起初，你是接收了整条消息后再转发它。你记录下发声器的嘀嗒，到消息接收结束，你再用你的按键将它们发送出去。最终你掌握了边听边发的技巧而不用把整条信息记录下来，这节约了转发时间。

某天你在转发消息时，你注意到铁棒上下跳动又注意到了手指按动键的上下跳动。你看了看发声器又看了看键，然后你意识到棒的上下跳动与按键的上下跳动是一致的，于是你出去取回一根小木条，用这根木条和一些线把发声器和按键连接了起来：

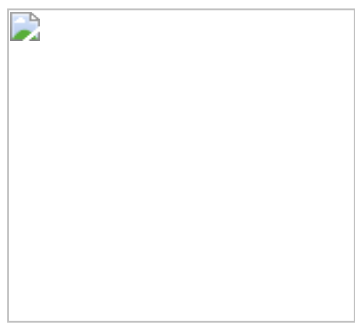


输出 输入

现在它可以自动工作了，你可以去喝下午茶也可以去钓鱼了。

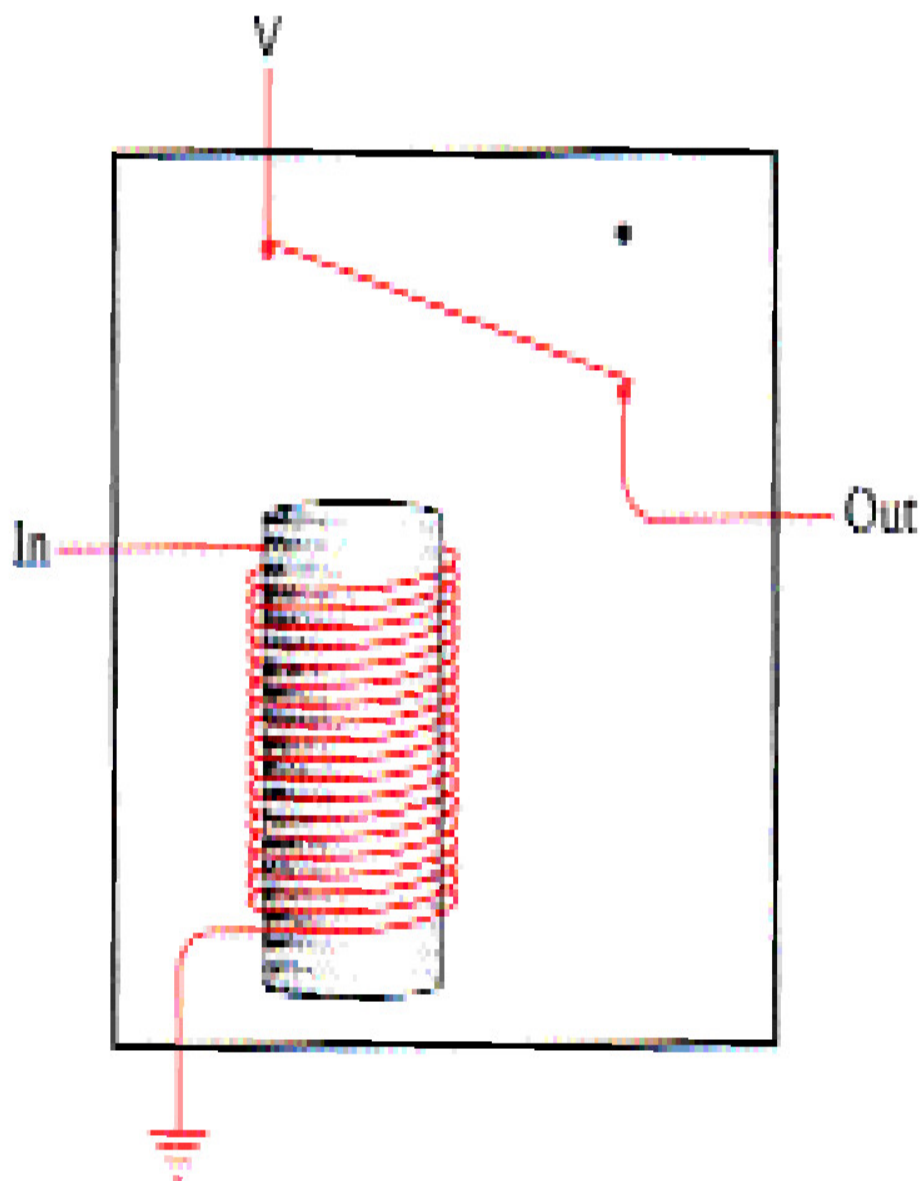
这只是一个趣味情景的想像。但实际上，摩尔斯很早就理解这个装置的思想。我们已经发明的这个装置叫重发器或继电器。一个继电器就像一个发声装置，输入的电流形成电磁用以拖动金属杆，金属杆作为开关的一个部分连接到外接的导线上。这样，微弱的输入电流被扩大形成比较强的输出电流。

继电器的概要描述如下图所示：



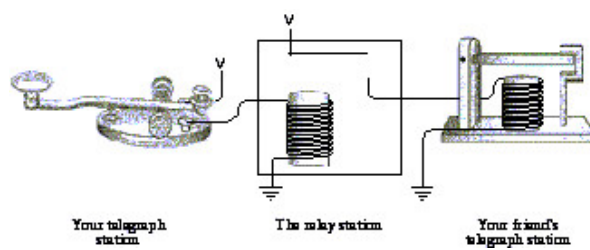
输入 输出

输入电流激发电磁铁，电磁铁吸引一根有弹性的金属条作为开关从而输出电流：



输入 输出

因此电报按键、继电器和发声器大致连接如下：



你的电报站 继电站 你朋友的电报站

继电器是一种卓越的设备。它是一个开关，但并不是由人工而是借助于电流进行开关操作的。利用这种设备可以做出令人惊奇的事情。事实上，你可以用继电器装配出一台计算机中的大部分部件。

是的，继电器这种设备是一种很好的发明，足以与电报相提并论。后面还将会用到，且它会变得非常小巧、方便。但是，在能够使用它之前，得先学会数数。

第 7 章 十进制记数法

语言仅仅是一种编码的想法似乎很容易被人们接受，很多人在学生时代至少学过一种外语，因此，我们知道在英语中“cat”（猫）也可以被叫作 gato、chat、Katze、KOIIIK 或 𐀀𐀁𐀂𐀃。

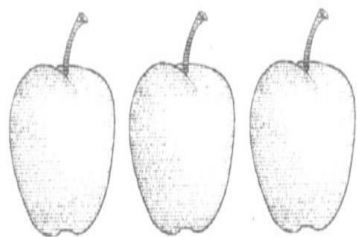
1 2 3 4 5 6 7 8 9 10

然而，数字不那么容易随文化的不同而改变。不论那种语言，也不管怎样读那些数字，地球上我们能够遇到的几乎所有的人都用同样的方式来写数字：

数学，从某种意义上来说是不是可以称得上是一种世界语言呢？毫无疑问，数字是我们平时能够接触到的最抽象的代码。当你看到数字“3”时并不需要



立即将它和任何事情相联系。你可能将它设想为 3 个苹果或者 3 个其他什么东西，但是当你从上下文中得知这个数字是指某个小孩的生日、电视频道、曲棍球比赛的得分或者是制作蛋糕的食谱中提供的需要面粉的杯数时，也能够像认为它代表 3 个苹果时一样自然。因为数字一开始产生时就很抽象，所以让我们理解这些苹果：



并不一定要用符号“3”来表示就更困难了。本章的很大一部分以及下一章将来讲解这些苹果：

也可以用“11”的形式来表示。

先不讨论数字 10 与生俱来的特殊性。大多数人使用的数字系统是基于 10（有时候是 5）的，这种情况并不奇怪。最初人们是用手指来数数的。要是人类进化成有 8 个或 12 个手指，人类计数的方式就会有所不同。英语 **Digit**（数字）这个单词也可以指手指或脚趾，单词 **five**（五）和单词 **fist**（拳头）有相同的词根，这种情况并不是巧合。

这样看来，人类选择使用以 10 为基础的记数方法（或称为十进制记数法）完全是任意的，但我们赋予 10 的整数次幂重大的意义，并给它们命名：十个一年是一个十年；十个十年是一个世纪；十个世纪是一个千年；千个一千是百万；千个百万是十亿。下面是 10 的各次幂：

$$10_1 = 10$$

$$10_2 = 100$$

$$10_3 = 1000 \text{ (千)}$$

$$10_4 = 10\,000$$

$$10_5 = 100\,000$$

$$10_6 = 1\,000\,000 \text{ (百万)}$$

$$10_7 = 10\,000\,000$$

$$10_8 = 100\,000\,000$$

$$10_9 = 1\,000\,000\,000 \text{ (十亿)}$$

多数历史学家认为数字最初创造出来是用来数东西的，比如：人数、财产数、商品交易量等。举个例子来说，假定某个人有 4 只鸭子，他可能画 4 只鸭子作为记录：



后来，专门负责画鸭子这项工作的人想：“我为什么一定要画 4 只鸭子呢？为什么不能只画 1 只鸭子，然后用其他方法（管它用什么方法，哪怕用一条竖线来代表一只鸭子）来表示有 4 只呢？”



但若某人有 27 只鸭子，用画竖线来表示鸭子只数的方法就显得很荒谬了：



于是，有人想到得有一种好的办法才行，数字系统就这样诞生了。在早期的数字系统中，只有罗马数字系统沿用至今。钟表的表盘上常常使用罗马数字，

此外它还用来在纪念碑或雕像上标注日期、标注书的页码，或作为提纲条目的标记。最令人惊奇的是罗马数字常用在电影中做版本说明。（只要有足够快的速度将字幕结尾处出现的 MCMLIII 译码，通常情况下就可以回答“这部影片是什么时候拍的”这个问题。）

27 只鸭子可以用罗马数字这样表示：



这里用到的概念非常简单：X 代表 10 条竖线，V 代表 5 条竖线。现在仍在使用的罗马数字有：

I V X L C D M

字母 **I** 代表一个一，这可能来自于一条竖线或者伸出的一个手指。字母 **V** 很可能是一只手的符号，代表五；两个字母 **V** 组成字母 **X**，代表十；字母 **L** 代表五十；字母 **C** 来自于拉丁文中表示一百的单词—**centum**；字母 **D** 代表五百；最后，字母 **M** 来自拉丁文中的单词—**mille**，代表一千。

也许你不一定同意，很长一段时间以来，罗马数字被认为用来做加减运算非常容易，这也是罗马数字能够在欧洲被长期用于记帐的原因。事实上，当对两个罗马数字进行相加运算时，只需将这两个罗马数字的所有符号合并然后用下面的方法将其简化：五个 **I** 是一个 **V**，两

个 **V** 是一个 **X**，五个 **X** 是一个 **L**，等等。但使用罗马数字做乘除法是很难的。很多其他早期的数字系统（比如古希腊数字系统）

和罗马数字系统相似，它们在做复杂运算时存在一定的不足。尽管如此，古希腊人所发明的

非凡的几何学至今仍是中学的一门课程，古希腊人不是以代数享誉世界的。我们现在使用的数字系统通常称为阿拉伯数字系统，或称为印度—阿拉伯数字系统。它

起源于印度，但由阿拉伯数学家传入欧洲。一位著名的波斯数学家—**Muhammed ibn-Musa al-Khwarizmi**（由它的名字得到单词 **algorithm**（算法））在大约公元 825 年写了一本代数书，书中用的就是印度的数字系统（阿拉伯数字）来计数。产生于公元 1120 年的拉丁文译本对整个欧洲用现在的阿拉伯数字代替当时使用的罗马数字的过渡过程产生了很大的影响。

印度—阿拉伯数字系统与先前的数字系统相比在以下三个方面不同：

- 印度-阿拉伯数字系统是和位置相关的，也就是说，一个数字依据位置的不同代表不同的数量。数字的位置和数字的大小一样，都是很重要的。（但实际上，数字的位置更重要。）100 和 1 000 000 中都只有一个 1，但我们知道一百万比一百要大得多。
- 几乎所有早期的数字系统都有一个阿拉伯数字所没有的东西，那就是用来表示数字 10 的一个专门的符号。现在使用的数字系统中是没有代表 10 的专门符号的。
- 另一方面，几乎所有早期的数字系统都缺少一个阿拉伯数字中有的，而且事实证明是比代表数字 10 的符号重要得多的符号，那就是零。是的，就是零。这个小小的零毫无疑问是数字和数学历史上最重要的发明之一。它支持

位置表示法，因为它可以将 205 与 250 区别开来。数字零也使得与位置无关的数字系统中非常复杂的运算变得简单，尤其是乘法。

印度—阿拉伯数字的整体结构是以读它们的方式展现的。拿 4825 作为例子，我们把它读作“四千八百二十五”，意思是：

或者，可以将它的组成写成这样：

四个一千 八个一百 两个十 一个五

$$4825=4000+800+20+5$$

或者，可以将它进一步分解，写成这样：

$$4825=4\times 1000+$$

$$8\times 100+$$

$$2\times 10+$$

$$5\times 1$$

另外，也可以使用 10 的整数次幂的形式，重新写成：

$$4825=4\times 10_3 +$$

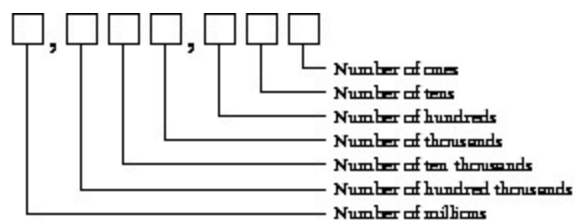
$$8\times 10_2 +$$

$$2\times 10_1 +$$

$$5\times 10_0$$

记住，任何数的 0 次幂都等于 1。

多位数中的每位都有特定的意义，如下图所示。这 7 个方格可以表示从 0~9 999 9999 的任何一个数字：



1 的个数

10 的个数

100 的个数

1000 的个数

10 000 的个数

100 000 的个数

1 000 000 的个数

每一个位置（位）与 10 的一个整数次幂相对应。不需要一个专门的符号来表示数字 10， 因为可以将 1 放在不同的位置，用 0 作为占位符。

分(小)数可以同样的形式作为数字放在十进制数的小数点的右边，这一点非常好。数字 42 705.684是：

$$4 \times 10\,000 +$$

$$2 \times 1000 +$$

$$7 \times 100 +$$

$$0 \times 10 +$$

$$5 \times 1 +$$

$$6 \div 10 +$$

$$8 \div 100 +$$

$$4 \div 1000$$

该数也可以写成不带除法的形式，如下：

$$4 \times 10\,000 +$$

$$2 \times 1000 +$$

$$7 \times 100 +$$

$$0 \times 10 +$$

$$5 \times 1 +$$

$$6 \times 0.1 +$$

$$8 \times 0.01 +$$

$$4 \times 0.001$$

或写成10的整数次幂的形式:

$$4 \times 10_4 +$$

$$2 \times 10_3 +$$

$$7 \times 10_2 +$$

$$0 \times 10_1 +$$

$$5 \times 10_0 +$$

$$6 \times 10_{-1} +$$

$$8 \times 10_{-2} +$$

$$4 \times 10_{-3}$$


注意10的指数是怎样变到零再变成负数的。

我们知道，3加上4等于7。同样，30加上40等于70，300加上400等于700，3000加上4000等于7000。这正是阿拉伯数字系统的“魅力”所在，无论你进行多长的十进制的加法，只要根据一种方法将问题分成几步即可。每一步最多只是将两个一位数字相加，这也是很久以前有人强迫你记加法表的原因：

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

从最上边的一行和最左边的一列找到要相加的两个数字，在行与列的交叉点上找到它们相加的结果。例如，4加上6等于10。

同样，做两个十进制数相乘的运算时，方法可能稍稍复杂一点儿，但仍然只需将问题分成几步，这样就不会比做加法和一位数的乘法更复杂了。你在小学时可能也必须记住下面的乘法表：



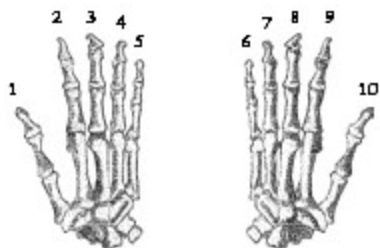
	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	14	16	18	20	22	24
3	3	6	9	12	15	18	21	24	27	30	33	36
4	4	8	12	16	20	24	28	32	36	40	44	48
5	5	10	15	20	25	30	35	40	45	50	55	60
6	6	12	18	24	30	36	42	48	54	60	66	72
7	7	14	21	28	35	42	49	56	63	70	77	84
8	8	16	24	32	40	48	56	64	72	80	88	96
9	9	18	27	36	45	54	63	72	81	90	99	108
10	10	20	30	40	50	60	70	80	90	100	110	120
11	11	22	33	44	55	66	77	88	99	110	121	132
12	12	24	36	48	60	72	84	96	108	120	132	144

与位置相关的记数系统的优点不在于它多么好用，而在于当它用在不是十进制的系统中时，也一样的好用。我们现在用的数字系统不一定适合所有的人。十进制数字系统的一个很大问题就在于它和卡通人物没有任何关系。大多数的卡通人物每只手上只有 4 个手指，因此它们喜欢基于 8 的数字系统（八进制）。有趣的是，我们所知的大部分关于十进制数的知识同样可以用于卡通朋友所喜爱的八进制数字系统中。

第 8 章 其他进位制记数 法

10对我们来说是一个非常重要的数字。10是我们大多数人拥有的手指或脚趾的数目，我

们当然希望所有人的手指脚趾都是 10 个。因为我们的手非常适合数数，因而我们人类已经适应了以10为基础的数字系统：



前面数章已经提到过，通常使用的数字系统称为以 10为基础的数字系统或十进制。这个数字系统对我们来说非常自然，因而我们很难想像出还有其他的数字系统。事实上，当我们看到数字 10的时候，不由自主地就会认为这个数是指下面这么多只鸭子：

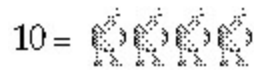
$$10 = \text{oooooooo}$$

但是，数字 10是指这么多只鸭子的唯一理由是因为这么多只鸭子与我们的手指数目相同。如果人类不是有那么多只手指，我们数

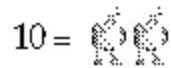
数的方式就会有所不同，数字 10 就可能代表别的东西了。同样是数字 10，可以指这么多只鸭子：



或这么多只鸭子：



甚至可以是这么多只鸭子：



当我们明白了 10 可以指只有两只鸭子的时候，也就可以解释开关、电线、灯泡、继电器

（或干脆就叫计算机）是怎样表示数字的了。

如果人类像卡通人物那样，每只手上只有 4 个手指会怎样呢？我们可能永远都不会想到要发明一种以 10 为基础的数字系统的问题，取而代之的是我们可能会认为数字系统基于 8 是正常、自然、合理、必然的，是毫无疑问的，是非常合适的。这时，就不能称之为十进制了，得将它称为以 8 为基础的数字系统或八进制。

9

如果数字系统是以 8 为基础组织起来的，就不需要这样的一个符号：

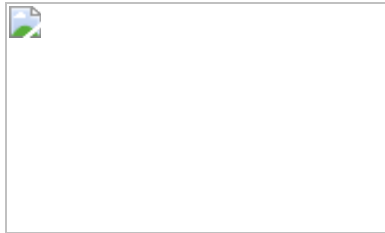
8

把这个符号拿给任何一个卡通人物看，都会有同样的反应：“那是什么？它是干什么用的？”如果再仔细想一会儿的话，你会发现连这样的一个字符也不需要：

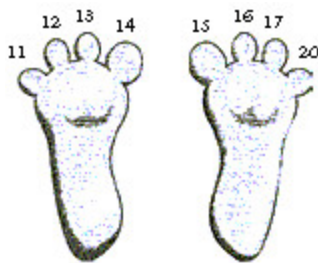
在十进制数字系统中，没有专门用来表示 10 的符号，而在八进制数字系统中，也没有专门用来表示 10 的符号。

在十进制数字系统中数数的方式是 0、1、2、3、4、5、6、7、8、9，然后是 10。在八进制数字系统中数数的方式是 0、1、2、

3、4、5、6、7，然后是什么呢？我们已经没有符号可用了，唯一的一个有意义的可用符号是 10，的确是那样。在八进制数中，7之后紧接着的数字是10，但是 10并不是指人类的手指那么多的数目。在八进制数中，10指的是卡通人物手指的数目：



继续数脚趾头：



使用非十进制的数字系统时，将数字“10”读作“么零”可以避免一些混淆。同样，“13”可以读作“么三”，“20”可以读作“二零”。要想真正避免混淆，可以将“20”读作“八进制 二零”或“基于 8 的数二零”。

即使没有手指和脚趾帮忙，我们仍能够将八进制数继续数下去。除了要跳过那些含有 8 或 9 的数字以外，它基本上和数十进制的数是一样的。当然，相同的数字代表的数量是不同的：

0、1、2、3、4、5、6、7、10、11、12、13、14、15、16、17、

20、21、22、23、24、25、26、27、30、31、32、33、34、35、
36、37、40、

41、42、43、44、45、46、47、50、51、52、53、54、55、56、
57、60、61、62、

63、64、65、66、67、70、71、72、73、74、75、76、77、100...

最后一个数字读作“么零零”，是卡通人物拥有的手指数自乘的结果（即平方）。在写十进制或八进制数时，为避免混淆，可以借助使用特定的标记以区别表示数字系统。

下面用标记“**TEN**”表示十进制数，标记“**EIGHT**”表示八进制数。

这样，白雪公主遇到的小矮人的数目是 7

TEN

或

EIGHT

卡通人的手指数是 8

10

TEN

或

EIGHT

贝多芬写的交响乐的首数是 9

11

TEN

或

EIGHT

人的手指的数目是 10

一年中的月份数是 12

12

TEN

或

14

或

TEN

EIGHT EIGHT

两个星期所包含的天数是 14

“情人”的生日庆祝会是 16

16

TEN

或

或20

EIGHT

一天中所包含的小时数是 24

EIGHT

或30

拉丁字母表中的字符数是 26

TEN

32

或

TEH

EIGHT

EIGHT

与一夸脱液体相当的盎司数为 32

40

TEN

或

EIGHT

一副牌中含有的牌数是 52

64

TEN

或

EIGHT

国际象棋棋盘的方格数是 64

100

TEN

或

EIGHT

Sunset Strip 最著名的 17 牌号是 77

or 115

EIGHT

美式足球场的面积是 100

144

TEN

或

EIGHT

参加温布尔登网球公开赛女单初赛的人数是 128

200

TEN

或

EIGHT

古埃及孟斐斯城市面积的平方英里数是 256

or 400

EIGHT

注意，在上面一系列的八进制数中，有一些好整数，像 100

EIGHT

、 200

EIGHT

、 400

EIGHT

◦ 好整

数通常是指结尾有一些零的数。在结尾处有两个零的十进制数意味着它是 100

即10

TEN

乘以

10

TEN

；在八进制数中，结尾处有两个零表示它是 100

EIGHT

即10

EIGHT

乘以 10

EIGHT

(或 8

乘以 8

等于 64

TEN

) °

你可能已经注意到了，好的八进制整数 100

200

EIGHT

、

400

EIGHT

和

EIGHT

与十进制数64

128

TEN

、

TEN

、

相等，它们都是 2 的整数次幂。例如， 400

EIGHT

等于

EIGHT

乘以10

EIGHT

乘以

EIGHT

，所有这些

数都是 2 的整数次幂。任何时候，将 2 的整数次幂和另一个 2 的整数次幂相乘，得到的仍是 2 的

TEN

整数次幂。

下表给出了一些 2 的整数次幂的十进制及其对应的八进制的表示形式：

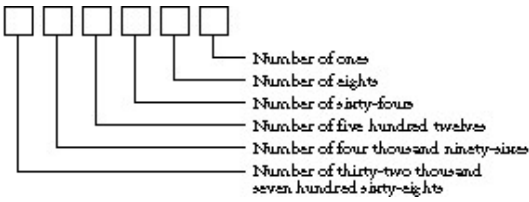
2 的整数次幂	十进制数	八进制数
2 ⁰	1	1
2 ¹	2	2
2 ²	4	4
2 ³	8	10
2 ⁴	16	20
2 ⁵	32	40
2 ⁶	64	100

(续)

2^7	128	200
2^8	256	400
2^9	512	1000
2^{10}	1024	2000
2^{11}	2048	4000
2^{12}	4096	10000

最右边一列的好整数给我们一个暗示：十进制以外的数字系统可能对你有帮助。

八进制数字系统和十进制数字系统在结构上没有什么差别，只是在细节上有一些差异。例如，八进制数的每一个位置代表的值是该位数字乘以 8 的整数次幂的结果：



1的个数

8 的个数

64 的个数

512 的个数

4096 的个数

32768 的个数

这样，八进制数 3725

EIGHT

可以拆分成这样：

EIGHT

$$= 3000$$

EIGHT

+ 700

EIGHT

+ 20

EIGHT

EIGHT

+ 5

还可以写成另外几种不同的形式。下面就是其中的一种，采用十进制形式的 8 的整数次幂：

EIGHT

$$= 3 \times 512$$

TEN

+

7×64

TEN

+

采用八进制形式的 8 的整数次幂的情况:

2×8

5×1

TEN

+

EIGHT

$$= 3 \times 1000$$

$$7 \times 100$$

EIGHT

+

+

还有另外的一种拆分形式:

2×10

5×1

EIGHT

EIGHT

+

EIGHT

$$= 3 \times 8_3 +$$

$$7 \times 8_2 +$$

$$2 \times 8_1 +$$

$$5 \times 8_0$$

如果算出其十进制的结果，会得到 2005

◦ 这就是将八进制数转换成十进制数的方法。

可以采用与做十进制加法和乘法相同的办法来做八进制数的加法和乘法。唯一真正的区别 在于要采用不同的表格来对各个数字进行乘法或加法运算。下面是八进制数的加法表：

TEN



例如, 5

EIGHT

EIGHT

+ 7

$$= 14$$

EIGHT

- 可以采用与做十进制加法相同的方法将两个稍长一点儿的

八进制数相加：

$$\begin{array}{r} 135 \\ + 643 \\ \hline 1000 \end{array}$$

先从最右边的一列做起，5加上3等于10，该位写下0，向前进1；1加3加4等于10，该位写下0，向前进1；1加1加6等于10。

同样，在八进制中，2乘以2仍然等于4。但是3乘以3却不等于9，那是多少呢？3乘以3等

EIGHT

，此数与 9

所代表的数量相等。下图是完整的八进制数的乘法表:

x	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

这里， 4×6 等于30

，也即表明 30

EIGHT

和 4×6 的十进制结果 24

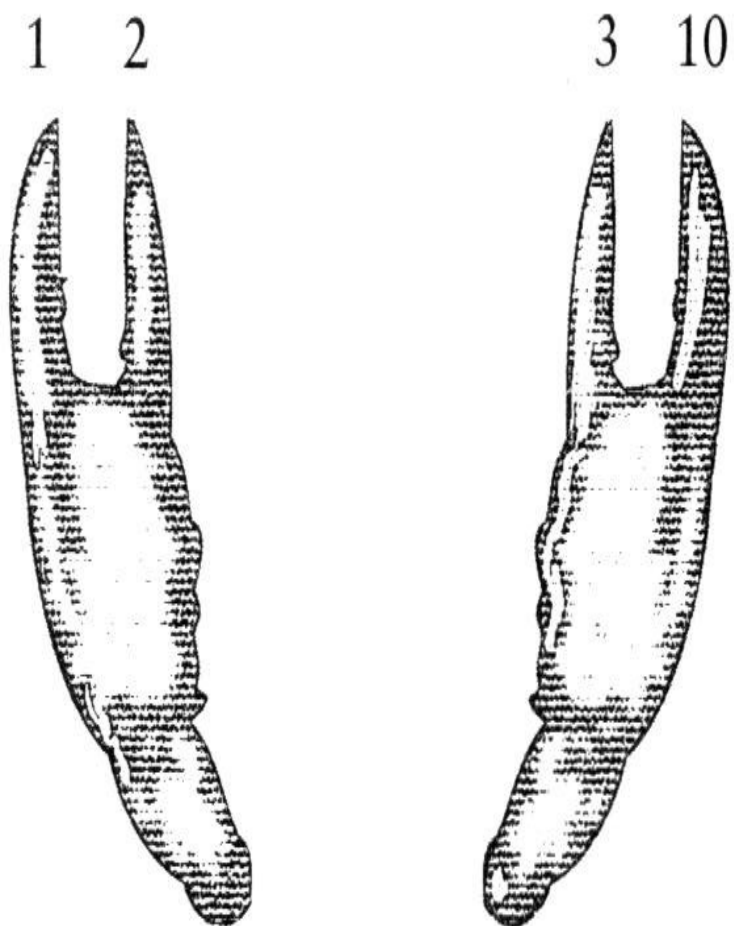
是等值的。

八进制数字系统与十进制数字系统一样，都是有效的，但八进制数字系统在理解上更深

EIGHT

TEN

了一层。既然我们已为卡通人物开发出了一套数字系统，就再给龙虾开发一套适合它们用的数字系统吧。龙虾根本没有手指，但它两只前爪的末端都有螯。适合于龙虾的数字系统是四



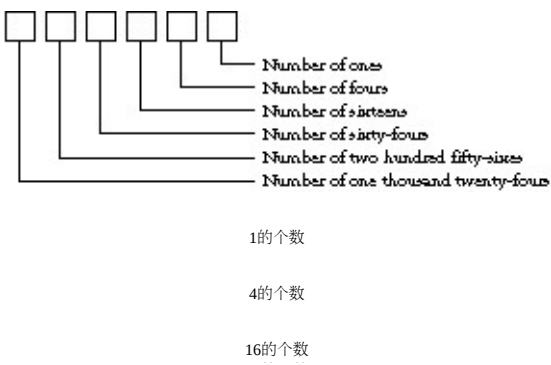
进制数字系统或称为基于 4 的数字系统:

四进制数可以这样来数：

0、1、2、3、10、11、12、13、20、21、22、23、30、31、32、33、100、101、102、

103、110，等等。这里不打算在四进制数上花太多的时间，因为还有更重要的事情要做。但我们还是要看

一下四进制中的每一位是怎样和 4 的某个整数次幂相对应的：



64的个数

256的个数

1024的个数

四进制数 31232 可以写成:

31232

FOUR

$$= 3 \times 256$$

$$1 \times 64$$

TEN

+

+

TEN

$$2 \times 16 \quad +$$

TEN

也可以写成:

3×4

2×1

TEN TEN

+

FOUR

$$= 3 \times 10000$$

+

FOUR

1×1000

FOUR

+

2×100

FOUR

+

3×10

2×1

FOUR

+

还可以写成:

31232

FOUR

FOUR

$$=3\times 4_4 +$$

$$1\times 4_3 +$$

$$2\times 4_2 +$$

$$3\times 4_1 +$$

$$2\times 4_0$$

如果以十进制数的形式计算其结果，就会发现 31232

FOUR

等于878

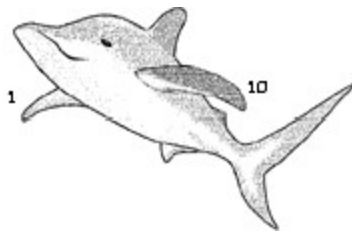
TEN

o

现在，我们要做一个跳跃并且是最远的一跳。假定我们是海豚，并且必须用两鳍来数数。

则这个数字系统就是基于 2 的数字系统或二进制的。这样似乎只需要两个数字，即 0 和 1。现在，0 和 1 已是你要处理的全部问题，需要练习一下才能习惯使用二进制数。二进制数

最大的问题是数字用完得很快。例如，下图是海豚怎样用它的鳍数数的例子：



是的，在二进制中，1 后面的数字是 10。这是令人惊讶的，但也并不奇怪。无论使用哪种数字系统，当单个位的数字用完时，第一个两位数字都是 10。在二进制系统中，可以这样来数数：

0、1、10、11、100、101、110、111、1000、1001、1010、
1011、1100、1101、1110、1111、10000、10001、……

这些数看起来好像很大，实际上并不是这样。更准确地说二进制数长度增长的速度要快过二进制数增大的速度：

每个人的头的个数为 1

TEN

或1

TWO

海豚身上的鳍的个数为 2

TEN

或10

TWO

一个大汤匙中包括的小茶匙的数目为 3

TEN

或11

TWO

正方形的边数为 4

或100

TWO

每个人一只手的手指数为 5

TEN

或101

TWO

一种昆虫的腿数为 6

TEN

一星期的天数为 7

或110

或111

TEN

TWO

TWO

八重奏中音乐家的个数为 8

或 1000

TWO

太阳系中的行星（包括冥王星在内）总数为 9

TEN

或1001

TWO

牛仔帽重量以加仑计算为 10

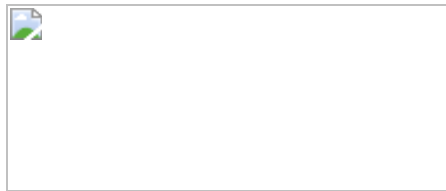
TEN

等等。

或1010

TWO

在多位二进制数中，数字的位置和 2 的整数次幂的对应关系为：



1 的个数

2 的个数

4 的个数

8 的个数

16 的个数

32 的个数

因此，任何时候由一个 1 后跟几个零构成的二进制数一定是 2 的整数次幂。2 的幂与二进制

数中零的个数相等。下面是扩充的 2 的各次幂的表，可用来说明这条规则：

2 的 幂	十进制数	八进制数	四进制数	二进制数
2 ⁰	1	1	1	1
2 ¹	2	2	2	10
2 ²	4	4	10	100
2 ³	8	10	20	1000
2 ⁴	16	20	100	10000
2 ⁵	32	40	200	100000
2 ⁶	64	100	1000	1000000
2 ⁷	128	200	2000	10000000
2 ⁸	256	400	10000	100000000
2 ⁹	512	1000	20000	1000000000
2 ¹⁰	1024	2000	100000	10000000000

211	2048	4000	200000	1000000000000
21 2	4096	10000	1000000	10000000000000

假定有一个二进制数 101101011010，它可以写成：

101101011010

TWO

$$= 1 \times 2048$$

TEN

+

0×1024

1×512

TEN

+

+

1×256

TEN

+

TEN

0×128 +

TEN

1×64

TEN

+

0×32

1×16

TEN

+

+

1×8

TEN

TEN

+

0×4

1×2

TEN

+

+

TEN

$$0\times 1$$

也可以这样写：

101101011010

TWO

TEN

$$= 1 \times 2_{11} +$$

$$0 \times 2_{10} +$$

$$1 \times 2_9 +$$

$$1 \times 2_8 +$$

$$0 \times 2_7 +$$

$$1 \times 2_6 +$$

$$0 \times 2_5 +$$

$$1 \times 2_4 +$$

$$1 \times 2_3 +$$

$$0 \times 2_2 +$$

$$1 \times 2_1 +$$

$$0 \times 2_0$$

如果将各个部分以十进制数的形式相加，得到
 $2048+512+256+64+16+8+2=2906$

TEN

o

将二进制数转换成十进制数非常简单，你可能更喜欢借助已准备好的模板进行转换：

--	--	--	--	--	--	--	--

$\times 128$ $\times 64$ $\times 32$ $\times 16$ $\times 8$ $\times 4$ $\times 2$ $\times 1$

	+		+		+		+		+		+		=	
--	---	--	---	--	---	--	---	--	---	--	---	--	---	--

× 64 × 32

× 16 × 8

这个模板允许你转换最大长度为 8 的二进制数，但它扩充起来非常容易。使用时，将 8 个二进制数字放到上部的 8 个小盒子中，一个盒子放一个数字。做 8 个乘法运算，将结果分别放

到底部的 8 个小盒子中。将 8 个盒子中的数字相加就得到最终结果。下面是将 10010110 转化成十进制数的例子：

1	0	0	1	0	1	1	0
---	---	---	---	---	---	---	---

x128	x64	x32	x16	x8	x4	x2	x1
------	-----	-----	-----	----	----	----	----

128	+	0	+	0	+	16	+	0	+	4	+	2	+	0	=	150
-----	---	---	---	---	---	----	---	---	---	---	---	---	---	---	---	-----

× 32 × 16 × 8 × 4 × 2 × 1

<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
÷128	÷64	÷32	÷16	÷8	÷4	÷2	÷1
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

将十进制数转换成二进制数就没那么直接了。但这里也有一个帮助你将 0~225范围内的 十进制数转换成二进制数的模板：

实际转化过程要表面上看的麻烦得多，所以一定要仔细按照下面的指导来做。将整个十进制数（应小于等于 225）放在左上角的方格中。用除数（128）去除那个数（被除数），如下图所示。将商写在正下方的盒子中（即左下角的盒子中），余数写在右边的盒子中（即上面一行左数第二个盒子中）。用第一个余数再除以下一个算子 64。依照模板的顺序用同样的方法继续做下去。

记住，每次求得的商只能是 0 或者 1。如果被除数小于除数，商为 0，余数和被除数相等；如果被除数大于除数，商为 1，余数为被除数与除数之差。下

面是将 150转换成二进制数的过 程：

150	22	22	22	6	6	2	0
÷128	÷64	÷32	÷16	÷8	÷4	÷2	÷1
1	0	0	1	0	1	1	0

如果要做两个二进制数的加法或乘法，也许直接采用二进制来做比转化成十进制再做还 要简单。这将是 你真正喜欢二进制数的地方。如果只需记住下面的二进制加法表就可以做加 法运算，也就不难想象掌握加法运算该有多快：

+	0	1
0	0	1
1	1	10

用二进制加法表将两个二进制数相加：

$$\begin{array}{r}
 1100101 \\
 + 0110110 \\
 \hline
 10011011
 \end{array}$$

从最右边的一列开始做起：1 加上 0 等于 1；右数第 2 列：0 加上 1 等于 1；第 3 列：1 加上 1 等于 0，进位为 1；第 4 列：1（进位值）加上 0 再加上 0 等于 1；第 5 列：0 加上 1 等于 1；第 6 列：1 加 1 等于 0，进位为 1；第 7 列：1（进位值）加上 1 再加上 0 等于 10。

乘法表比加法表更简单，因为该表可以由两个基本的乘法规则推导出来：零乘以任何数 都等于 0，1 与任何数相乘仍是那个数本身：

×	0	1
0	0	0
1	0	1

下面是13

TEN

与11

以二进制数的形式做乘法的过程:

1 1 0 1

TEN

× 1 0 1 1

1 1 0 1

1 1 0 1

0 0 0 0

1 1 0 1

1 0 0 0 1 1 1 1

最后结果是 143

TEN

o

人们在使用二进制数的时候通常将它们写成带有前导零的形式（即第一个 1 的左边有零）。

例如，0011 而不写成 11。这些零不会改变数字的值，只是起到一些装饰作用。例如，下面是二进制的前 16 个数以及和它们等值的十进制数：

二进制数	十进制数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11

1100

12

1101

13

1110

14

1111

15

让我们再仔细看看这些二进制数字。考虑一下这 4 个垂直列中每一列的 0 和 1，注意它们在

一列中自上而下是以怎样的规律变化的：

- 最右边一列一直在 0和1之间相互替换。
- 右数第2列在两个 0和两个1之间相互替换。
- 右数第3列在四个 0和四个1之间相互替换。
- 右数第4列在八个 0和八个1之间相互替换。这是很有规律的，难道不是吗？事实上，只要再重复这 16个数字并且在每个数字的前面

放一个1就可以很容易地写出后面的 16个数字：

二进制数	十进制数
10000	16
10001	17
10010	18
10011	19
10100	20
10101	21
10110	22

10111	23
11000	24
11001	25
11010	26
11011	27
11100	28
11101	29
11110	30
11111	31

下面是看待这些数字的另一种方式：在数二进制数的时候，最右边的数字（也称最低位 数字）是在 0和1之间变化的。当它每次从1变到0时，右数第二位数字（也称次低位数字）也要发生变化，或者从 0变到1，或者从 1变到 0。每次只要有一个二进制数位的值由 1变到0，紧挨着的高位数字也会发生变化，要么从 0变到1，要么从 1变到0。

我们在写十进制中比较大的数字时，通常每三个数字之间留一点儿空隙，这样，我们一看就知道这个数的大概数值。例如，当你看到数字 12000000时，你可能不得不去数其中 0的个数，但如果看到的是 12 000 000，则马上就能知道是一亿两千万。

二进制数的位长度增加得特别快。例如，一亿两千万的二进制表示为：101101110001101100000000。为了让它更易读，通常是每四个数字之间用连字符或空格来分开。例如；1011-0111-0001-1011-0000-0000或101101110001101100000000。本书的后面会讲

到更简单的二进制数的表示方法。

通过将数字系统减少至只有 0 和 1 两个数字的二进制数字系统，我们已经在能够接受的范围内做了深入的讨论。不可能找到比二进制数字系统更简单的数字系统了。二进制数字系统架起了算术与电之间的桥梁。前面各章中，我们所看到的开关、电线、灯泡、继电器等物体都可以表示二进制数 0 和 1：

电线可以表示二进制数字。有电流流过电线代表二进制数字 1；如果没有，则代表二进制数字 0。

开关可以表示二进制数字。如果开关闭合，代表二进制数字 1；如果开关断开，代表二进制数字 0。

灯泡可以表示二进制数字。如果灯泡亮着，代表二进制数字 1；如果没亮，代表二进制数

字0。

电报继电器可以表示二进制数字。继电器闭合，代表二进制数字1；继电器断开，代表二进制数字0。

二进制数与计算机密切相关！

大约在1948年，美国数学家 John Wilder Tukey（生于1915年）提前认识到二进制数将在未来几年中随着计算机的流行而发挥更大的作用。他决定创造一个新的、更短的词来代替使用起来很不灵活的五音节词—**binary digit**。他曾经考虑用 **bigit**或**binit**，但最后还是选用了短小、简单、精巧且非常可爱的单词 **bit**(比特)来代替 **binary digit**这个词。

第9章 二进制数

1973年，当安东尼·奥兰多在他写的一首歌中要求他挚爱的人“系一条黄色的绸带在橡树上”时，他并没有要求他的爱人进行繁琐的解释或冗长的讨论，只要求她给他一个简单的结果。他不去关心其中的因果，即使歌中复杂的感情和动情的历史在现实生活中重演，所有的人真正想知道的仅仅是一个简单的是或不是。他希望在树上系一条黄色的绸带来表示：“是的，即使你犯了很大的错，并且被判了入狱三年，我仍希望你回来和我一起共渡时光。”他希望用树上没有黄色的绸带来表示：“你连停在这里都别想。”

这是两个界线分明、相互排斥的答案。奥兰多没有这样唱：“如果你想再考虑一下的话，就系半条黄色的绸带”或者“如果你不爱我但仍希望我们是朋友，就系一条蓝色的绸带吧”。相反，他让答案非常的简单。

和黄色绸带的有无具有同样效果的另外几个例子（但可能无法用在诗里）是可以选择一种交通标记放在院外，可能是“请进”或“此路不通”。

或者在门上挂一个牌子，上写“关”或“开”。或者用从窗口能够看到的一盏灯的亮灭来表示。如果你只需说“是”或“不是”的话，可以有很多种方式来表达。你不必用一个句子来

表达是或不是，也不需要一个单词，甚至连一个字母都不要。你只要用一个比特，即只要一个0或1即可。

正如我们在前面的章节中所了解到的，通常用来计数的十进制数事实上并没有什么与众不同的地方。非常清楚，我们的数字系统之所以是基于10的（十进制数）是因为我们有10个手指头。我们同样有理由使用八进制数字系统（如果我们是卡通人物）或四进制数字系统

（如果我们是龙虾），甚至是二进制数字系统（如果我们是海豚）。但是，二进制数字系统有一点儿特别：它可能是最简单的数字系统。二进制数字系统中

只有两种二进制数字—0和1。要是我们想寻求更简单的数字系统，只好把数字1去掉，这样，就只剩下0一个数字了。只有一个数字0的数字系统是什么都做不成的。

“bit(比特)”这个词被创造出来代表“binary digit”，它的确是新造的和计算机相关的最可爱的词之一。当然，“bit”有其通常的意义：“一小部分，程度很低或数量很少”。这个意义用来表示比特是非常精确的，因为1比特—一个二进制数字位—确实是一个非常小的量。

有时候当一个新词诞生时，它还包含了一种新的意思。bit这个词也是这样。1比特的意思超过了被海豚用来数数的二进制数字位所包含的意义。在计算机时代，比特已经被看作是组成信息块的基本单位。

当然，上述说法不一定完全正确，比特并不是传送信息的唯一的方式。字母、单词、摩尔斯码、布莱叶盲文，十进制数字都可以用来传递信息。比特传递的信息量很小。1比特只具备最少的信息量，更复杂的信息需要多位比特来传递。（我们说比特传递的信息量小，并不是说它传送的信息不重要。事实上黄绸带对于与它相关的两个人来说是一个非常重要的信息。）“听，孩子们，你们很快就能听到 Paul Revere 午夜的马蹄声。”亨利·朗费罗写道。尽管

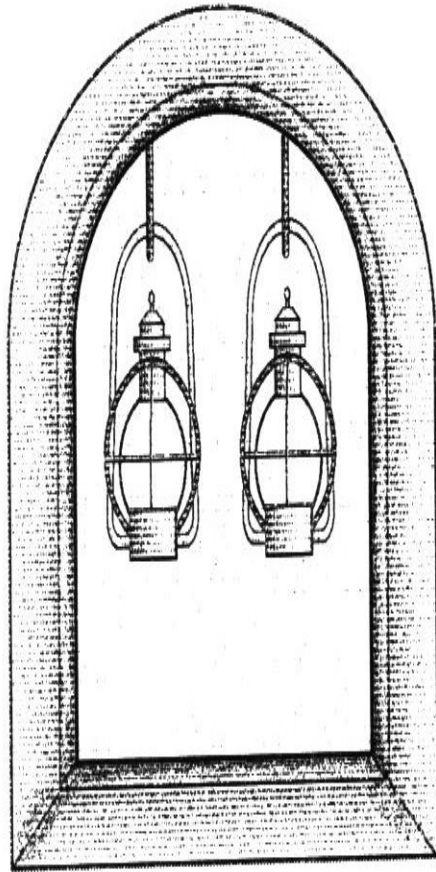
*He said to his friend "If the British march
By land or sea from the town to-night,
Hang a lantern aloft in the belfry arch
Of the North Church tower as a special light,—
One, if by land, and two, if by sea..."*

他在描述 Paul Revere 是怎样通知美国人英国殖民者入侵的消息时不一定与史实完全一致，但 他的确提供了一个利用比特传递信息的令人茅塞顿开的例子：

(他告诉他的朋友：“如果英军今晚入侵， 你就在北教堂的钟楼拱门上悬挂点亮的提灯 作为信号。一盏提灯代表英军由陆路入侵， 两盏提灯代表英军由海路入侵。……)

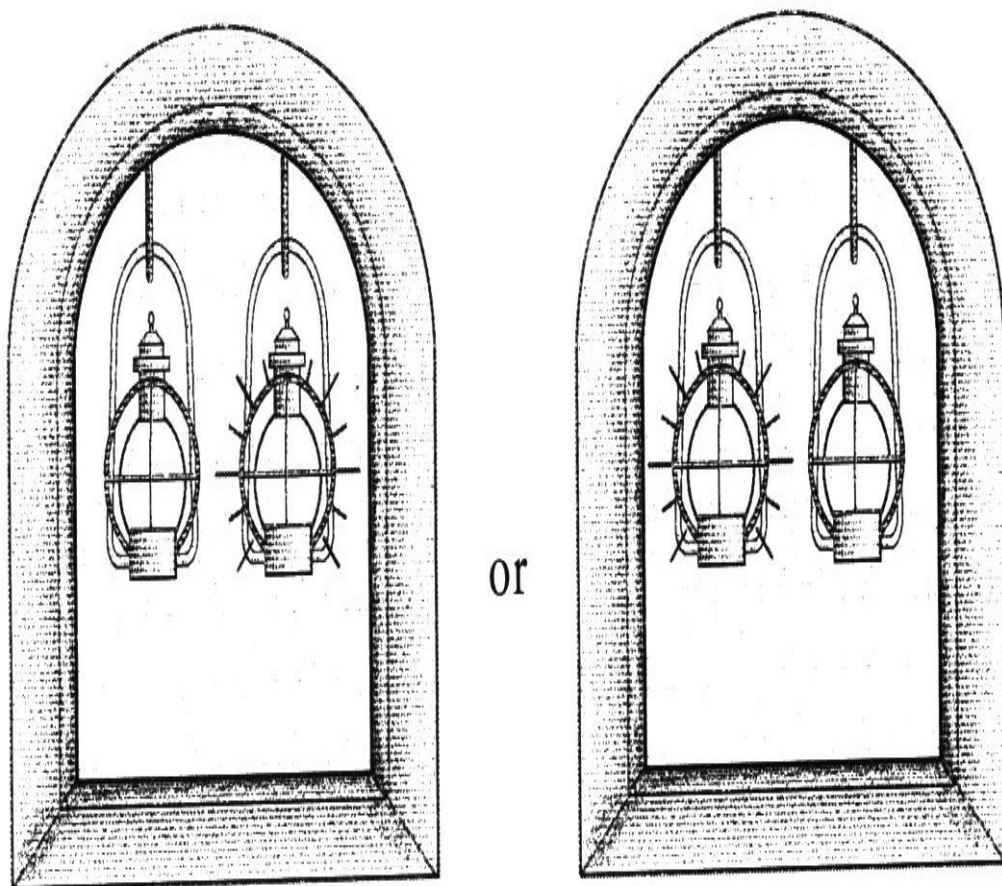
也就是说，Paul Revere 的朋友有两盏灯。如果英军由陆路入侵，他就挂一盏灯在教堂的钟楼上；如果英军由海路入侵，他就挂两盏灯在教堂的钟楼上。

然而，朗费罗并没有将所有的可能都涉及到。他留下第三种情况没有说，那就是英军根本就没有入侵的情况。朗费罗已经暗示第三种可能的信息可以由不挂提灯的方式来传递。



让我们假设那两盏灯是永久固定在教堂钟楼上的。在正常情况下，它们都不亮：

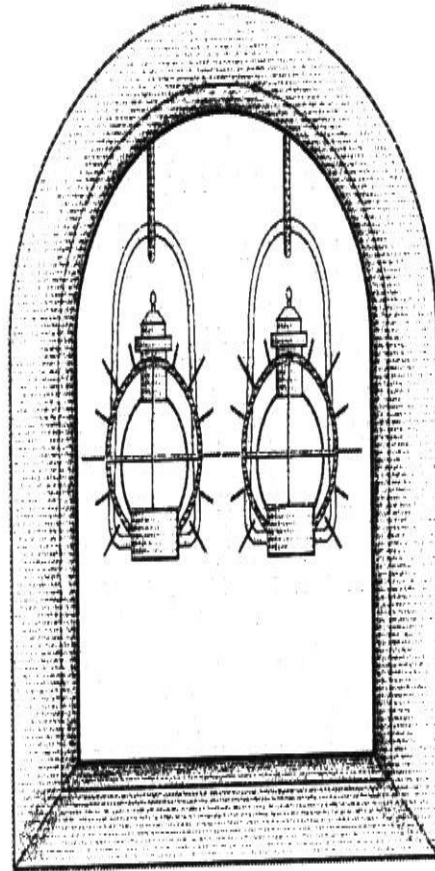
这就是指英军还没有入侵。如果一盏提灯亮：



or

或

表示英军正由陆路入侵。如果两盏提灯都亮：



表示英军正由海路入侵。

每一盏提灯都代表一个比特。亮着的灯表示比特值为 1，未亮的灯表示比特值为 0。前面 奥兰多已经说明了传送只有两种可能性的信息只需要一个比特。如果 **Paul Revere** 只需被告知 英军正在入侵（不管是从何处入侵）的消息，一盏提灯就足够了。点亮提灯代表英军入侵，未点亮提灯代表又是一个和平之夜。

传递三种可能性的消息还需要再有一盏提灯。一旦再有一盏提灯，两个比特就可以通知 有四种可能的信息：

00=英军今晚不会入侵

01=英军正由陆路入侵

10=英军正由陆路入侵

11=英军正由海路入侵

Paul Revere 将三种可能性用两盏提灯来传送的做法事实上是相当富有经验的。用通信理论的术语说，他采用了冗余的办法来降低噪声的影响。通信理论中的噪声是指影响通信效果的任何事物。电话线路中的静电流显然是影响电话通信的一种噪声。然而，即使是在有噪声的情况下，电话通信仍能够成功，因为口语中存在大量的冗余。你同样可以听懂对方的话而无需将每个音节、每个字都听得很清楚。

在上述例子中，噪声是指晚上光线黯淡以及 **Paul Revere** 距钟楼有一定的距离，它们都阻碍了 **Paul Revere** 声将钟楼上的两盏灯区分清楚。下面是朗费罗的诗中很重要的一段：

*And lo! As he looks, on the belfry's height
A glimmer, and then a gleam of light!
He springs to the saddle, the bridle he turns,
But lingers and gazes, till full on his sight
A second lamp in the belfry burns!*

（哦！他站在与钟楼等高的位置观察，一丝微光，然后，有一盏灯亮了！他跳上马鞍，调转马头，徘徊，凝视，直到看清所有的灯 另一盏灯也亮了！）

那当然不是说 Paul Revere正在辨清到底是哪盏灯先亮的问题。这里最本质的概念是信息可能代表两种或多种可能性的一种。例如，当你和别人谈话时，

说的每个字都是字典中所有字中的一个。如果给字典中所有的字从 1 开始编号，我们就可能精

确地使用数字进行交谈，而不使用单词。（当然，对话的两个人都需要一本已经给每个字编过号的字典以及足够的耐心。）

换句话说，任何可以转换成两种或多种可能的信息都可以用比特来表示。不用说，人类使用的很多信息都无法用离散的可能性来表示，但这些信息对我们人类的生存又是至关重要

的。这就是人类无法和计算机建立起浪漫关系的原因所在（无论怎样，都希望这种情况不会

发生）。如果无法将某些信息以语言、图片或声音的形式表达，那也不可能将这些信息以比特的形式编码。当然，你也不会想将它们编码。

举手或不举手是一个比特的信息。两个人是否举手——就像电影评论家 Roger Ebert和刚去

世不久的 **Gene Siskel**对新影片提供他们最终的评价结果那样——传递两个比特的信息。（我们将忽略掉他们实际上对影片做的评语，而只关心他们有没有举手的问题。）这样，我们用两个 比特代表四种可能：

00 = 他们都不喜欢这部影片

01 = Siskel 讨厌它， Ebert喜欢它 10 = Siskel 喜欢它， Ebert讨厌它 11 = Siskel和Ebert都喜欢它

第一个比特值代表 **Siskel**的意见， 0表示 **Siskel**讨厌这部影片， 1表示 **Siskel**喜欢这部影片。同样，第二个比特值代表 **Ebert**的意见。

因此，如果你的朋友问你 **Siskel**和**Ebert**是怎么评价《**Impolite Encounter**》这部电影的，你

不用回答“**Siskel**举手了，**Ebert**没有举手”或者“**Siskel**喜欢这部电影，**Ebert**不喜欢这部电影”，你可以简单地回答“么零”。你的朋友只要知道哪一位代表的是 **Siskel**的意见，哪一位代表的是**Ebert**的意见，并且知道值为 1代表举手，值为 0代表没有举手，你的回答就是可以被人理解的。当然，你和你的朋友都要知道这种代码的含义。

我们也可以一开始就声明值为 1的比特位表示没有举手，值为 0的比特位表示举手了，这

可能有点违反常规。通常我们会认为值为 1的比特位代表正面的事情，而值为 0的比特位代表相反的一方面，这的确只是一种很随意的指派。无论怎样，用此种代码的人只要明白 0、1分别代表什么就可以了。

某一位或几位比特位的集合所代表的意义通常是和上下文相关的。橡树上的黄绸带可能只有系绸带的人和期望看到绸带的人知道其中的意思，改变绸带的颜色、系绸带的树或系绸带的日期，绸带可能会被认为只是一块毫无意义的破布。同样，要从 Siskel 和 Ebert 的手势中得到有用的信息，我们至少要知道正在讨论的是哪部影片。

如果你保存了 Siskel 和 Ebert 对一系列影片的评价和投票结果，你就有可能在表示 Siskel 和

Ebert 意见的比特信息中再增加一位代表你自己的观点的比特位。增加的第三位使得其代表的信息可能性增加到 8 种：

000 = Siskel 讨厌它， Ebert 讨厌它， 我讨厌它 001 = Siskel 讨厌它， Ebert 讨厌它， 我喜欢它 010 = Siskel 讨厌它， Ebert 喜欢它， 我讨厌它 011 = Siskel 讨厌它， Ebert 喜欢它， 我喜欢它 100 = Siskel 喜欢它， Ebert 讨厌它， 我讨厌它 101 = Siskel 喜欢它， Ebert 讨厌它， 我喜欢它 110 = Siskel 喜欢它， Ebert 喜欢它， 我讨厌它 111 = Siskel 喜欢它， Ebert 喜欢它， 我喜欢它

使用比特来表示信息的一个额外好处是我们清楚地知道我们解释了所有的可能性。我们知道有且仅有 8 种可能性，不多也不少。用 3 个比特，我们只能从 0 数到 7，后面再没有 3 位二进制数了。

在描述 Siskel 和 Ebert 的比特时，你可能一直在考虑一个严重的，并且是令人烦恼的问题

——对于 Leonard Maltin 的 Movie & Video Guide 怎么办呢？别忘了，Leonard Maltin 是不采用举

手表决这种形式的，他对电影的评价用的是更传统的星级系统。

要想知道需多少个 **Maltin** 比特，首先要了解一些关于 **Maltin** 评分系统的知识。**Maltin** 给电影的评价是 1~4 颗星，并且中间可以有半颗星。（仅仅是为了好玩，他实际上不会给电影只评一颗星，取而代之的是给一个 **BOMB** [炸弹]。）这里总共有七种可能性，也就是说只需要 3 个比特位就可以表示一个特定的评价等级了：

000 = BOMB

001 = ★1/2

010 = ★★

011 = ★★1/2

100 = ★★★

101 = ★★★1/2

110 = ★★★★

你可能会问 111 怎么办呢，111 这个代码什么意义都没有，它没有定义。如果二进制代码 111 被用来表示 **Maltin** 等级，那一定是出现错误了。（这可能是计算机出的错误，因为人不会给出这样的评分。）

前面我们曾用两个比特来代表 **Siskel** 和 **Ebert** 的评价结果，左边的一位代表 **Siskel** 的评价意见，右边的一位代表 **Ebert** 的评价意见。在上述 **Maltin** 评分系统中，各个比特位都有确定的意义吗？是的，当然有。将比特编码的数值加 2 再除以 2，就得到了 **Maltin** 评

分中对应的星的颗数。这样编码是由于我们在定义代码时遵循了合理性和连贯性，我们也可以下面的这种方式编码：

000=★★★

001=★1/2

010=★★1/2

011=★★★★

101=★★★1/2

110=★★

111=BOMB

只要大家都了解代码的含义，这种表示就和前述代码一样，都是合理的。如果Maltin遇到了一部连一颗星都不值得给的电影，他就会给它半颗星。他当然有足够的

代码来表示半颗星的情况，代码会像下面这样定义：

000=MAJOR BOMB

001=BOMB

010=★1/2

011=★★

100=★★1/2

101=★★★

110=★★★1/2

111=★★★★

但是，如果他再遇到连半颗星的级别都不够的影片并且决定给它没有星的级别（**ATOMIC BOMB?**），他就得再需要一个比特位了，已经没有 3 个比特的代码空闲了。

《**Entertainment Weekly**》杂志常常给事物定级，除了电影之外还有电视节目、**CD**、书籍、**CD-ROM**、网络站点等等。等级的范围从 **A+~F**，如果你数一下的话，发现共有 13 个等级。这样，需要四个比特来代表这些等级：

0000 = F

0001 = D-

0010 = D

0011 = D+

0100 = C-

0101 = C

0110 = C+

0111 = B-

1000 = B

1001 = B+

1010 = A-

1011 = A

1100 = A+

有3个代码没有用到，它们是：1101、1110和1111，加上后总共是16个代码。只要谈到比特，通常是指特定数目的比特位。拥有的比特位数越多，可以传递的不同可

能性就越多。

对十进制数当然也是同样的道理。例如，电话号码的区号有几位呢？区号共有3位数字。

如果所有的区号都使用的话（实际上有一部分区号并没有使用，将它们忽略），一共有 10^3 或

1000个代码，从000~999。区号为212的7位数的电话号码有多少种可能呢？ 10^7 或10 000 000

个；区号为212并且以260开头的电话号码有多少个呢？ 10^4 或10 000个。同样，在二进制数中，可能的代码数等于2的比特位数次幂：

比特位数	代码数
------	-----

1	$2^1 = 2$
---	-----------

2	$2^2 = 4$
---	-----------

3	$2^3 = 8$
---	-----------

4	$2^4 = 16$
---	------------

5	$2^5 = 32$
---	------------

6	$2^6 = 64$
---	------------

7	$2^7 = 128$
---	-------------

8	$2^8 = 256$
---	-------------

9	$2^9 = 512$
---	-------------

每增加一个比特位，二进制代码数翻一番。如果知道需要多少个代码，那么怎样才能知道需要多少个比特位呢？换句话说，在上述

表中，如何才能由代码数反推出比特位数呢？

用到的方法叫作取以 2 为底的对数，对数运算是幂运算的逆运算。我们知道 2 的 7 次幂等于

128，以2为底的128的对数就等于 7。用数学记号来表示第一个句子为：

$$2_7 = 128$$

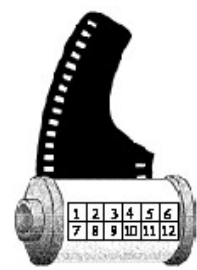
它与下述句子等价：

$$\log 128 = 7$$

因此，如果以 2 为底的 128 的对数等于 7，以 2 为底的 256 的对数等于 8，那么，以 2 为底的 200 的对数等于多少呢？大约是 7.64，但实际上并不需要知道它。如果要表示 200 种不同的事物，我们共需要 8 个比特。

比特通常无法从日常观察中找到，它深藏于电子设备中。我们看不到压缩磁盘 (CD)、数字手表或计算机中编过码的比特，但有时候比特也可以清晰地看到。

下面就是一个例子。如果你手头有一个使用 35 毫米胶片的相机，观察一下它的卷轴。这样拿住胶卷：



胶卷上有像国际跳棋棋盘一样的银色和黑色方格，方格已用数字 1~12 标识。这叫作 **DX** 编码，这 12 个方格实际上是 12 个比特。一个银色的方格代表值为 1 的比特，一个黑色的方格代表值为 0 的比特。方格 1 和 7 通常是银色的（代表 1）。

这些比特是什么意思呢？你可能知道有些胶片对光的敏感程度要比其他胶片强，这种对光的敏感程度称作 胶片速度。说对光非常敏感的胶片很快是因为这种胶片的曝光速度快。曝光速度是由 **ASA**（**American standards association**，美国标准协会）来制定等级的，最常用的等级有 100、200 和 400。ASA 等级不只是以十进制数字的形式印在胶卷的外包装和暗盒上，而且还以比特的形式进行了编码。

胶卷总共有 24 个 ASA 等级，它们是：

25	32	40
50	64	80
100	125	160
200	250	320

400

500

640

800

1000

1250

1600

2000

2500

3200

4000

5000

为ASA等级编码需要多少个比特呢？答案是 5 个比特。我们知道， $2_4 = 16$ ，与 24 比太小了； $2_5 = 32$ ，又超过了所需的编码数。

比特值与胶片速度的对应关系如下所示:

方格2	方格3	方格4	方格5	方格6	胶片速度
0	0	0	1	0	25
0	0	0	0	1	32
0	0	0	1	1	40
1	0	0	1	0	50
1	0	0	0	1	64
1	0	0	1	1	80
0	1	0	1	0	100
0	1	0	0	1	125
0	1	0	1	1	160
1	1	0	1	0	200
1	1	0	0	1	250
1	1	0	1	1	320

0	0	1	1	0	400
0	0	1	0	1	500
0	0	1	1	1	640
1	0	1	1	0	800
1	0	1	0	1	1000
1	0	1	1	1	1250
0	1	1	1	0	1600
0	1	1	0	1	2000
0	1	1	1	1	2500
1	1	1	1	0	3200
1	1	1	0	1	4000
1	1	1	1	1	5000

多数现代的 35毫米照相机胶片用的都是这些代码（除了那些要手工进行曝光的相机和具有内置式测光表但需要手工设置曝光速度的相机以外）。如果你看过照相机的内部放置胶卷的地方，你应该能够看到和胶片的金属方格（1~6号）相对应的 6个金属可接触点。银色方格实际上是胶卷暗盒中的金属，是导体；油漆了的黑色方格，是绝缘体。

照相机的电子线路中有一支流向方格 1 的电流，方格 1 通常是银色的。这支电流有可能流到方格 2~6，这要依方格中是纯银还是涂了油漆而定。这样，如果照相机在接触点 4 和 5 检测到了电流而在接触点 2、3 和 6 没有检测到，胶片的速度就是 400ASA。照相机可以据此调节曝光时间。

廉价的照相机只要读方格 2 和方格 3，并且假定胶片速度是 50、100、200 或 400ASA 四种可能速度之一。

多数相机不读方格 8~12。方格 8、9、10 用来对这卷胶卷进行编码；方格 11 和 12 指出曝光范围，依胶片用于黑白照片、彩色照片还是幻灯片而定。

也许最常见的二进制数的表现形式是无处不在的 UPC (universal product code, 通用产品代码)，即日常所购买的几乎所有商品包装上的条形码。条形码已经成为计算机在日常生活中应用的一种标志。

尽管 UPC 常常使人多疑，但它确实是一个无辜的小东西，发明出来仅仅是为了实现零售业的结算和存货管理的自动化，且其应用是相当成功的。当它和一个设计精良的结算系统共同使用时，顾客可以拿到列出细目的售货凭条，这一点是传统现金出纳员所无法做到的。

有趣的是，UPC 也是二进制代码，尽管它初看起来并不像。将 UPC 解码并看看 UPC 码具

体是怎样工作的是很有益的。

通常情况下，UPC是30条不同宽度的垂直黑色条纹的集合，由不同宽度的间隙分割开，

3

其下标有一些数字。例如，以下是 Campbell公司10 盎司的罐装鸡汁面包装上的 UPC:

4



可将条形码形象地看成是细条和黑条，窄间隙和宽间隙的排列形式，事实上，这是观察条形码的一种方式。黑色条有四种不同的宽度，较宽的条的宽度是最细条的宽度的两倍、三倍或者四倍。同样，各条之间的间隙中较宽的间隙是最窄间隙的两倍、三倍或者四倍。



但是，看待 UPC的另一种方式是将它看作是一系列的比特。记住，整个条形码与条形码扫描仪在结算台“看”到的并不完全一样。扫描仪不会识别条形码底部的数字，因为识别数字需要一种更复杂的技术——光学字符识别技术，又称作 OCR (optical character recognition)。实际上，扫描仪只识别整个条形码的一条窄带，条形码做得很大是为了便于结算台的操作人员用扫描仪对准顾客选购的物品。扫描仪所看到的那一条窄带可以这样表示：

它看上去是不是很像摩尔斯编码？

当计算机自左向右进行扫描时，它给自己遇到的第一个条分配一个值为 1 的比特值，给与 条相邻的间隙分配一个值为 0 的比特值。后续的间隙和条被当作一行中一系列比特中的 1 个、2 个、3 个还是 4 个比特读进计算机要依据条或间隙的宽度而定。扫描进来的条形码的比特形式很简单：

因此，整个UPC只是简单的由95个比特构成的一串。本例中，这些比特可以像下面这样分组：



101000110101100010011001000110100011010001101000110101010111001011001101101100100111011001101000100101

Bits	Meaning
101	Left-hand guard pattern
0001101	Left-side digits
0110001	
0011001	
0001101	
0001101	
0001101	
01010	Center guard pattern
1110010	Right-side digits
1100110	
1101100	
1001110	
1100110	
1000100	
101	Right-hand guard pattern

比特 意义

最左边的护线

左边的数字

中间的护线

右边的数字

最右边的护线

起初的3个比特通常是 101，这就是最左边的护线，它帮助计算机扫描仪定位。从护线中，扫描仪可以知道代表单个比特的条或间隙的宽度，否则，所有包装上的 UPC印刷大小都是一样的。

紧挨着最左边的护线是每组有 7个比特位的六组比特串，每一组是数字 0~9的编码之一，我们在后面将证明这一点。接着的是 5个比特的中间护线，此固定模式（总是 01010）是一种内置式的检错码。如果扫描仪在应当找到中间护线的地方没有找到它，扫描仪就认为那不是 UPC。中间护线是防止条形码被窜改或错印的方法之一。

中间护线的后面仍是每组 7个比特的 6组比特串。最后是最右边的护线，也总是 101。最后的最右护线使得 UPC反向扫描（也就是自右向左扫描）同正向扫描一样成为可能，这一点我们将在后面解释。

因而整个 UPC对12个数字进行了编码。左边的 UPC包含了 6个数字的编码，每个数字占有 7个比特位。你可以用下表进行解码：

左边的编码

0001101=0	0110001=5
0011001=1	0101111=6
0010011=2	0111011=7
0111101=3	0110111=8
0100011=4	0001011=9

注意，每个 7位代码都是以 0开头，以 1结尾的。如果扫描仪遇到了第一个比特位值为 1或最后一个比特位值为 0的情况，它就知道自己没有将 UPC正确地读入或者是条形码被窜改了。另外我们

还注意到每个代码都仅有两组连续的值 1 的比特位，这就意味着每个数字对应着条形码中的两个竖条。

上表中的每个代码中都包含有奇数个值为 1 的比特位，这也是用于检测差错和数据一致性的一种机制，称为奇偶校验。如果一组比特位中含有奇数个 1，就称之为奇校验；如果含有偶数个 1，就称之为偶校验。这样看来，所有这些代码都拥有奇校验。

为了给 UPS 右边的 7 位一组的数字解码，可以采用下面的表格：

右边的编码

1110010=0	1001110=5
1100110=1	1010000=6
1101100=2	1000100=7
1000010=3	1001000=8
1011100=4	1110100=9

这些代码都是前述代码的补码或补数：凡是 1 的地方都换成 0，凡是 0 的地方都换成 1。这些代码都是以 1 开始，以零结束，并且每组都有偶数个 1，称之为偶校验。

现在，可以对 UCP 进行解码了。借助前两个表格，Campbell 公司 10 3 盎司的罐装鸡汁面

的包装上用 **UPC**编码的12个数字是:

0 51000 01251 7

这个结果是令人失望的，正如你所看到的那样，它们和印在 UPC 底部的数字完全相同。

（这样做是有意义的，因为由于某种原因，扫描仪可能无法识别条形码，收银员就可以手工将这些数字输进去。）我们还没有完成解码的全部任务，而且，我们也无法从中解码任何秘密信息。然而，关于 UPC 的解码工作已经没有了，那 30 个竖条已经变成了 12 个数字。

第一个数字（在这里是 0）被称为数字系统字符，0 的意思是说这是一个规范的 UPC 编码。如果是具有不同重量的货物的 UPC（像肉类或其他商品），这个数字是 2；订单、票券的 UPC

编码的第一个数字通常是 5。

紧接着的 5 个数字是制造商代码。在上例中，51000 是 Campbell 鸡汁面公司的代码。

Campbell 公司生产的所有产品都使用这个代码。再后面的 5 个数字（01251）是该公司的某种

3

产品的编号，上例中是指 10 4 盎司的罐装鸡汁面。别的公司的鸡汁面可能有不同的编号，且

01251 在另外一个公司可能是指一种完全不同的产品。

和通常的想法相反，UPC 中没有包含该种产品的价格。产品的价格信息可以从商店中使用的与该扫描仪相联的计算机中检索互到。

最后的数字（这里是 7）称作模校验字符，这个字符可用来进行另外一种错误检验。为了解释校验字符是怎样工作的，将前 11 个数字（是 0 51000 01251）各用一个字母来代替：

A B C D E F G H I J K

然后，计算下式的值：

$$3\times (A+C+E+G+I+K) + (B+D+F+H+J)$$

从紧挨它并大于等于它的一个10的整倍数中减去它，其结果称为模校验字符。在上例中，有：

$$3 \times (0+1+0+0+2+1) + (5+0+0+1+5) = 3 \times 4 + 11 = 23$$

紧挨23并大于等于 23的一个10的整倍数是 30,故：

$$30 - 23 = 7$$

这就是印在外包装上并以 UPC形式编码的模校验字符，这是一种冗余措施。如果扫描仪 计算出来的模校验结果和 UPC中编码中的校验字不一致，计算机就不能将这个 UPC作为一个 有效值接收。

正常情况下，表示从 0~9的十进制数字只需 4个比特就足够了。在 UPC中，每个数字用了 7个比特，这样总共有 95个比特来表示 11个有用的十进制数字。事实上， UPC中还包括空白位置（相当于 9个0比特），位于左、右护线的两侧。因而，总共有 113个比特用来编码 11个十进制数，平均每个十进制数所用超过了 10个比特位！

正像我们所知道的那样，有部分冗余对于检错来讲是必要的。这种商品编码如果能够很

容易地被顾客用粗头笔修改的话，这种代码措施也就难以发挥其作用了。

UPC编码可以从两个方向读，这一点是非常有益的。如果扫描仪解码的第一个数字是偶 校验（即：每 7位编码中共有偶数个 1），扫描仪就知道它正在从右向左进行解码。计算机系统 用下表对右边的数字解码：

逆向时右边数字的代码

$$0100111 = 0 \quad 0111001 = 5$$

$$0110011 = 1 \quad 0000101 = 6$$

$$0011011 = 2 \quad 0010001 = 7$$

$$0100001 = 3 \quad 0001001 = 8$$

$$0011101 = 4 \quad 0010111 = 9$$

下面是对左边数字的解码表:

逆向时左边数字的代码

$$1011000 = 0 \quad 1000110 = 5$$

$$1001100 = 1 \quad 1111010 = 6$$

$$1100100 = 2 \quad 1101110 = 7$$

$$1011110 = 3 \quad 1110110 = 8$$

$$1100010 = 4 \quad 1101000 = 9$$

这些 7 位编码与扫描仪由左向右扫描时所读到的编码完全不同，但不会有模棱两可的现象。

让我们再看看本书中提到的由点、划组成其间用空格分开的摩尔斯电码。摩尔斯电码看上去不像是由 0 和 1 组成的，但它确实是。

下面回忆一下摩尔斯电码的编码规则：划的长度等于点长度的三倍；单个的点或划之间用长度与点的长度相等的空格分开；单词内的各个字母之间用长度等于划的长度的空格分隔；各单词之间由长度等于两倍的划长度的空格分开。

为使分析更加简单，我们假设划的长度是点长度的两倍而不是 3 倍。也就是说，一个点是一个值为 1 的比特位，一个划是两个值为 1 的比特位，空格是值为 0 的比特位。

下面是第 2 章的摩尔斯电码的基本表：

A	•—	J	•—•—•—	S	••••
B	•••••	K	—••—	T	—•
C	••••••	L	•—•••	U	•••••
D	—•••	M	—•—•	V	••••••
E	•	N	—••	W	••••••
F	•••••	O	—•—•—	X	—•••••
G	—•—•	P	•—•—•	Y	—••—•—
H	••••	Q	—•••••	Z	—•—•••
I	••	R	•—•••		

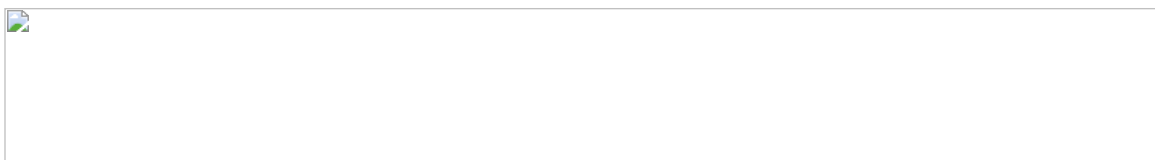
下面是将它转化为比特形式的结果:

A	101100	J	101101101100	S	1010100
B	1101010100	K	110101100	T	1100
C	11010110100	L	1011010100	U	10101100
D	11010100	M	1101100	V	1010101100
E	100	N	110100	W	101101100
F	1010110100	O	1101101100	X	11010101100
G	110110100	P	10110110100	Y	110101101100
H	101010100	Q	110110101100	Z	11011010100
I	10100	R	10110100		



注意，所有的编码都以 1 开头，以两个 0 结束。结尾处的两个零代表单词中各个字母之间的空格，单词之间的空格用另外的一对 0 来表示。因而，“Hi,there”的摩尔斯电码通常是这样的：

但是，采用比特形式的摩尔斯电码看起来像 UPC 编码的横切面：



用比特的形式表示布莱叶盲文比表示摩尔斯电码容易得多。布莱叶编码是 6 比特代码。布莱叶盲文中的每一个字母都是由 6 个点组成的，点可能是凸起的，或没有凸起（平滑）的。如在 第 3 章中讲的那样，这些点通常用数字 1~6 编号：

1 ○ ○ 4
2 ○ ○ 5
3 ○ ○ 6



例如，单词“code”可以用布莱叶盲文这样表示：

如果突起的点是 1，平坦的点是 0，则布莱叶盲文中的每一个符号都可以用 6 个比特的二进制代码表示。单词“code”中的四个布莱叶字母符号就可以简单地写成：

100100 101010 100110 100010

最左边的一位对应编号为 1 的位置，最右边的一位对应编号为 6 的位置。正如前面所讲到的，比特可以代表单词、图片、声音、音乐、电影，也可以代表产品编

码、胶片速度、电影的受欢迎程度、英军的入侵以及某人所挚爱的人的意愿。但是，最基本的一点是：比特是数字。当用比特表示信息时只要将可能情况的数目数清楚就可以了，这样就决定了需要多少个比特位，从而使得各种可能的情况都能分配到一个编号。

比特在哲学和数学的奇怪混合物——逻辑——中发挥作用。逻辑最基本的目标是证明某个语句是否正确，正确与否也可以用 1 和 0 来表示。

第 10 章 逻辑与开关

真理是什么呢？亚里士多德认为逻辑与它有关。他的讲义合集《工具论》（**Organon**，可追溯到公元前 4 世纪）是最早的关于逻辑的详细著作。对于古希腊人而言，逻辑是追寻真理的过程中用于分析语言的一种手段，因此它被认为是一种哲学。亚里士多德的逻辑学的基础是三段论。最有名的三段论（它并非是在亚里士多德的著作中发现的）是：



（所有的人都是要死的；苏格拉底是人；所以，苏格拉底是要死的。）

在三段论中，两个前提被假设是正确的，并由此推出结论。苏格拉底之死这个例子看上去似乎太直白了，但还有许多其他不同的三段论。例如，考

*All philosophers are logical;
An illogical man is always obstinate.*

虑下面两个由 19 世纪数学家 Charles Dodgson（也就是 Lewis Carroll）提出的前提：

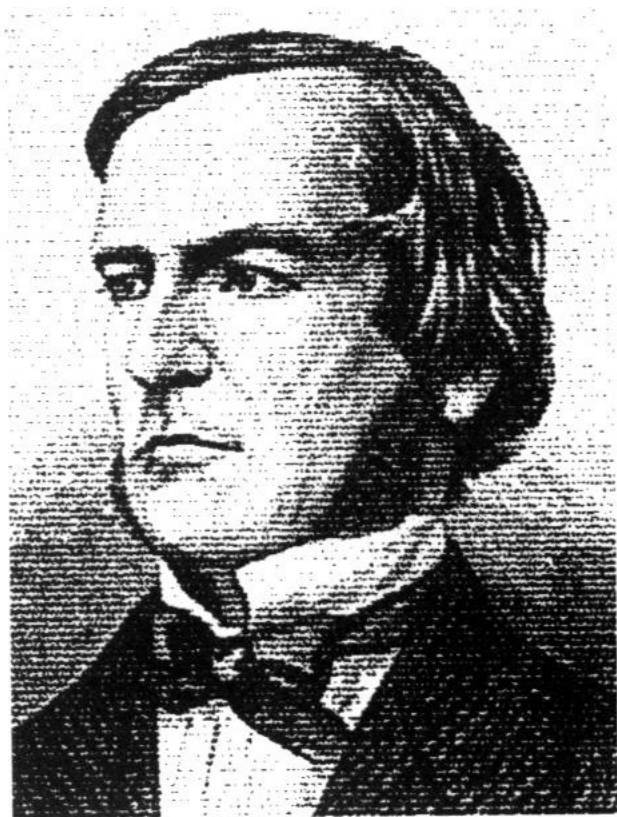
（所有的哲学家都是有逻辑头脑的；

一个没有逻辑头脑的人总是顽固的。）

它所能推出的结论一点儿也不明显。（事实上，结论是“一些顽固的人不是哲学家 (Some obstinate persons are not philosophers)”）请注意结论中一个出乎意料且令人迷惑的词“一些 (some)”。

两千多年来，数学家们对亚里士多德的逻辑理论苦苦思索，试图用数学符号和操作符来表现它。19 世纪以前，唯一能接近这个目标的人是莱布尼兹（1648—1716），他早年涉足逻辑学领域，后来转

向其他学科（比如说，他几乎和牛顿同时独立地发明了微积分）。



接下来有所突破的是乔治·布尔。

乔治·布尔 1815年生于英格兰，他周围的环境对他的成长很不利。他父亲是鞋匠，而母亲曾是女仆，英国森严的等级制度使布尔学不到什么有别于父辈的东西。但是靠着他自己强烈的好奇心及父亲的帮助（其父对科学研究 数学和文学有浓厚的兴趣），年轻的乔治自学了上层阶级 男孩才能学到的课程，包括拉丁文、希腊语及数学。由于他早年在数学方面发表的论文，1849年，布尔被任命为爱尔兰Cork市的皇后大学数学系的首席教授。

19世纪中期的几位数学家在逻辑理论的数学定义上做了一些工作（最著名的是迪摩根），但只有布尔有真正概

念上的突破。他最早的贡献是发表的一本很简短的书《The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning》(1847)，接着又发表了一篇很长且

充满抱负的文章：《An Investigation of the Laws of Thought on Which Are Founded the

Mathematical Theories of Logic and Probabilities》(1854)，简称为《The Laws of Thought》。1864年的一天，布尔在雨中赶去上课时不幸感染上了肺炎，不治身亡，享年 49 岁。

我们可以从布尔在 1854 年所著书的题目中看出他富于野心的想法：由于充满理性的人脑

用逻辑去思考，那么，如果能用数学来表征逻辑，我们也就可以用数学来描述大脑是如何工作的。当然，现在看来这种想法似乎十分幼稚。（但却超越了他所在的年代。）

布尔发明了一种和传统代数看起来、用起来都十分相似的代数。在传统代数中，操作数

（通常是字母）代表数字，而操作符（多是“+”或“×”）指明这些操作数如何结合到一起。一般我们可用传统代数解决类似下面的问题：如果安娜有 3 磅豆腐，贝蒂的豆腐是安娜的 2

倍，卡门的豆腐比贝蒂多 5 磅，迪尔德丽的豆腐是卡门的 3 倍。那么，迪尔德丽有多少豆腐呢？

为了计算这个问题，我们首先把语句转化为算术式子，用四个字母代表每个人拥有豆腐的数量，即：

$$A = 3$$

$$B = 2 \times A \quad C = B + 5$$

$$D = 3 \times C$$

可以通过代入把上述四个表达式合为一个式子，最后执行加法和乘法，即：

$$D = 3 \times C$$

$$D = 3 \times (B+5)$$

$$D = 3 \times ((2 \times A) + 5) \quad D = 3 \times ((2 \times 3) + 5) \quad D = 33$$

当做传统代数题时，要遵循一定的规则。这些规则可能已经和实践融为一体，以至于我们不再认为它们是规则，甚至忘记了它们的名字。但规则确实是任何形式的数学的基础。

第一个规则是加法与乘法的交换律，即我们可以在操作符两边交换操作数的位置：

$$A+B = B+A \quad A \times B = B \times A$$

相反，减法和除法是不满足交换律的。加法和乘法也满足结合律，即：

最后，乘法对加法可以进行分配：

$$A + (B + C) = (A + B) + C \quad A \times (B \times C) = (A \times B) \times C$$

$$A \times (B + C) = (A \times B) + (A \times C)$$

传统代数的另外一个特点是它总是处理数字，如豆腐的重量或鸭子的数量，火车行驶的距离或家庭成员的年龄。是布尔超凡的智慧使代数脱离了数字的概念而变得更加抽象。在布尔代数中（布尔的代数最终被这样命名）操作数不是指数字，而是指集（类）。一个类仅仅表示一组事物，也就是后来熟知的集合。

让我们来讨论一下猫。猫或公或母，为方便起见，我们用字母 **M** 指代公猫的集合，用 **F** 指

代母猫的集合。记住，这两个符号并不代表猫的数量，公猫或母猫的数量随着小猫仔的出生和老猫的不幸离去而变化，这两个字母代表的是猫的种类——具有某种特点的猫。因而我们不说公猫，而是用 **M** 来代表它们。

我们也可以用其他字母代表猫的颜色。例如，用 **T** 代表黄褐色的猫，用 **B** 代表黑猫，用 **W**

代表白猫，而用 **O** 代表所有其他颜色的猫。最后（至少就这个例子而言），猫要么是阉过的要么是有生育能力的。我们用字母 **N** 代表

阉过的猫，而用 **U** 代表有生育能力的猫。

在传统代数中，操作符 $+$ 和 \times 被用于表示加法和乘法。在布尔代数中，同样用到了 $+$ 和 \times 。这似乎会引起混淆。人人都知道在传统代数中如何对数字进行加和乘，但是我们如何对“类”进行加和乘呢？

事实上，在布尔代数中我们并不真正地做加或乘，相反，这两个符号有着完全不同的意思。

在布尔代数中，符号 $+$ 意味着两个集合合并，两个集合的合并就是包含第一个集合的所有

成员及第二个集合的所有成员。例如， $B+W$ 表示黑猫和白猫的集合。布尔代数中的符号 \times 意味着取两个集合的交集，两个集合的交集包含的元素既在第一个

集合中，也在第二个集合中。例如， $F \times T$ 代表了一种猫的集合，这个集合中的猫既是母猫又是黄褐色的。与传统代数一样，我们可以把 $F \times T$ 写成 $F \cdot T$ 或简写为 FT （这正是布尔代数所期望的）。你可以把这两个字母看成是连在一起的两个形容词：黄褐色的母猫。

为避免传统代数和布尔代数之间的混淆，有时候用符号 \cup 和 \cap 而不用 $+$ 和 \times 来表示并运算和交运算。但布尔对数学的解放性的部分影响是使熟悉的操作符更加抽象，所以，我们决定

坚持他的决定，而不为他的代数引入新的符号。交换律、结合律和分配律在布尔代数中均适用。而且，在布尔代数中，操作符 $+$ 可以对 \times

进行分配，这在传统代数中是不成立的，即：

$$W + (B \times F) = (W + B) \times (W + F)$$

这个式子表示白猫（ W ）和黑色母猫（ $B \times F$ ）的并集和等式右边两个集合的交集是一样的，这两个集合是白猫和黑猫的并集（ $W + B$ ）及白猫和母猫的并集（ $W + F$ ）。要掌握这个规则有些困难，但它的确有用。

为了使布尔代数更加完整，我们还需要两个符号。这两个符号看上去像数字，但它们并不真的是数字，因为有时候它们和数字有些不同。符号“1”在布尔代数中表示“整个宇宙

(全集)”，也就是我们所谈论的每件事物。本例中，符号“1”表示“所有的猫”。这样：

$$M + F = 1$$

即母猫和公猫的并集是所有的猫。同样，黄褐色猫、黑猫、白猫及其他颜色的猫的并集也是所有的猫，即：

你也可以这样表示所有的猫:

$$T+B+W+O=1$$

$$N+U=1$$

符号1可以用一个减号—来排除一些事物。例如：

$$1-M$$

表示除了公猫以外的所有猫。排除公猫以后的全集就是母猫的集合：

$$1-M = F$$

我们所需要的另外一个符号是“0”。在布尔代数中，“0”表示空集，即不含任何事物的集合。当求取两个完全相互排斥的集合的交集时，空集就产生了。例如，既是母的又是公的猫的集合可以表示为：

$$F \times M = 0$$

注意，符号1和0有时的用法与传统代数相同。例如，所有的猫和母猫求交集即是母猫这个集合：

空集和母猫求交集还是空集： 空集和母猫的并是母猫这个集合：

$$1 \times F = F$$

$$0 \times F = 0$$

$$0 + F = F$$

但有时与传统代数中得到的结果就不太一样了。例如，所有的猫和母猫的并集是所有的 猫：

$$1 + F = 1$$

这个表达式在传统代数中是没有意义的。

由于F代表母猫的集合， $1 - F$ 代表所有其他猫的集合，则这两个集合的并集是 1：

$$F + (1 - F) = 1$$

并且它们的交集是 0 :

$$F \times (1 - F) = 0$$

历史上，这个公式代表了逻辑中一个十分重要的概念，即矛盾律。它表明一个事物不能同时是它自己和它自己的反面。

使布尔代数和传统代数看起来完全不同的是下面这个表达式：

$$F \times F = F$$

这个式子在布尔代数中有着完美的意义：母猫的集合和母猫的集合的交集仍旧是母猫的集合。但若 F 代表一个数字的话，这个公式显然就不对了。布尔认为：

$$X_2 = X$$

是使他的代数与传统代数区分开来的唯一表达式。另一个按照传统代数看起来很有趣的布尔表达式是：

母猫和母猫的并集仍是母猫这个集合。

$$\mathbf{F} + \mathbf{F} = \mathbf{F}$$

布尔代数为解决亚里士多德的三段论提供了一个数学方法。再看这个著名三段论的两个前提：

所有的人都是要死的； 苏格拉底是人。

我们用字母 **P** 代表所有人的集合， **M** 代表要死的东西的集合， **S** 代表苏格拉底。那么，所谓“所有的人都是要死的”意味着什么呢？它其实表示了所有人的集合和所有要死的东西的集合的交集是所有人这个集合，即：

$$P \times M = P$$

而 $P \times M = M$ 这个式子是错误的，因为要死的东西还包括猫、狗、榆树等等。而“苏格拉底是人”意味着苏格拉底这个集合（非常小）和所有人的集合（很大）的交

集是苏格拉底这个集合：

$$S \times P = S$$

由于从第一个式子中知道 $P = P \times M$ ，所以可以把它代入第二个式子，即：

$$S \times (P \times M) = S$$

根据结合律，上式等同于：

$$(S \times P) \times M = S$$

但我们已经知道 $S \times P$ 等于 S ，所以上式可简化为：

$$S \times M = S$$

现在计算完毕。这个表达式告诉我们，苏格拉底和所有要死东西的集合的交集是苏格拉底，也就是说苏格拉底是要死的。相反，如果认为 $S \times M$ 等于 0 ，那么结论就是苏格拉底不会死。再如果，若 $S \times M$ 等于 M ，则能推出的结论就是苏格拉底是唯一会死去的东西，而其他任何东西都是不朽的！

用布尔代数来证明显而易见的事实似乎有些小题大做（尤其当考虑到苏格拉底早已在 2400 年以前就去世了时），不过，布尔代数还可以用来判断一些事物是否满足一定的标准。也许有一天，你走进宠物店对店员说：“我想要一只阉过的公猫，白的或黄褐色的均可；或者要一只没有生殖能力的母猫，除了白色，其他任何颜色均可；或者只要是只黑猫，我也要。”店员对你说：“看来您想要的猫是下面的式子表示的集合中的一只：

$(M \times N \times (W+T)) + (F \times N \times (1-W)) + B$ 对吗？”你回答道：“是的，完全正确！”为了证明店员是正确的，你可能想放弃并和交的概念而转向“OR（或者 / 或）”和“AND

（并且 / 与）”。大写这两个词是因为虽然在通常情况下它们代表语言中的概念，但它们也代表了布尔代数中的操作。当求两个集合的并集时，你实际上是从第一个集合“或”从第二个集

合中取得事物放入结果集合里。当求两个集合的交集时，满足条件的事物必定在第一个集合

中“并且”也在第二个集合中。此外，每当你看见后跟减号的 1，你可以使用单词“NOT（非）”

来表示。小结如下：

- +（以前表示求并集）现在表示 OR。
- ×（以前表示求交集）现在表示 AND。
- 1-（以前表示从全集中排除一些事物）现在表示 NOT。这样，刚才的表达式可以写成下面的形式：

$$(M \text{ AND } N \text{ AND } (W \text{ OR } T)) \text{ OR } (F \text{ AND } N \text{ AND } (\text{NOT } W)) \text{ OR } B$$

这与你的口头描述已经十分接近了。注意圆括号是如何清楚地表达出你的意图的。你想要的猫来自下面三个集合之一：

$$(M \text{ AND } N \text{ AND } (W \text{ OR } T))$$

或

$$(F \text{ AND } N \text{ AND } (\text{NOT } W))$$

或

$$B$$

写下这个公式后，店员就可以进行布尔测试的工作了。别这么大惊小怪的，这里已经悄悄转移到另一种不同形式的布尔代数中去了。在这种形式的布尔代数中，字母不再只表示集合，字母还可以被赋予数字，但需要注意的是它们只能被赋予 0 或者 1。数字 1 表示“是的”、

“正确”，本例中的意思是“这只猫符合我的要求”；数字 0 表示“否定”、“错误”、本例中即“这只猫不符合我的要求”。

首先，店员拿出一只未阉过的黄褐色的公猫。下面是满足条件的猫的集合：

$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$ 当用0和1代替字母后就变成了下面的样子：

$(1 \times 0 \times (0 + 1)) + (0 \times 0 \times (1 - 0)) + 0$ 注意被赋予了1的字母只有M和T，因为拿来的这只猫是公的，黄褐色的。

现在必须要做的是简化这个表达式。如果简化后表达式的结果是1，这只猫就满足了你的

要求，否则就不是你想要的猫。当简化表达式时，千万记住我们并不是在真正地做加法和乘法。当+表示OR，×表示AND时，大部分规则是相同的。（现代课本中有时用Λ和V分别表示AND和OR，而不用×和+；但这里用+和×这两个符号却是恰到好处的。）

当用×表示AND时，可能的结果是：

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

换句话说，只有当×的左、右两个操作数均为1时，结果才为1。这个过程和普通乘法一模一样。若用一张小表总结一下，你会发现它们和第8章的加法表和乘法表的形式相似：

AND	0	1
0	0	0
1	0	1

当用+表示OR时，可能的结果是：

$$0+0 = 0$$

$$0+1 = 1$$

$$1+0 = 1$$

$$1+1 = 1$$

当+的左、右操作数中有一个为 1 时，结果就是 1。除了 1+1=1 这种情况，这种计算和普通加法产生的结果是一致的。可用另一张小表来总结：

OR	0	1
0	0	1
1	1	1

现在可以用这些表来计算前面那个表达式的结果了：

$$(1 \times 0 \times 1) + (0 \times 0 \times 1) + 0 = 0 + 0 + 0 = 0$$

结果是0，表示“否定”、“错误”，即这只小猫不满足客户需求。接下来，店员拿来一只无生育能力的白色的小母猫。原始表达式是：

$$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$$

把0和1代入上式： 并且把它简化一下：

$$(0 \times 1 \times (1+0)) + (1 \times 1 \times (1-1)) + 0$$

$$(0 \times 1 \times 1) + (1 \times 1 \times 0) + 0 = 0 + 0 + 0 = 0$$

看来，这只可怜的小猫还是不符合要求。然后，店员又拿来一只无生育能力的灰色的小母猫。（灰色是非白色、黑色或黄褐色的
一

种其他颜色。）下面是表达式：

$$(0 \times 1 \times (0+0)) + (1 \times 1 \times (1-0)) + 0$$

现在把它简化为:

$$(0 \times 1 \times 0) + (1 \times 1 \times 1) + 0 = 0 + 1 + 0 = 1$$

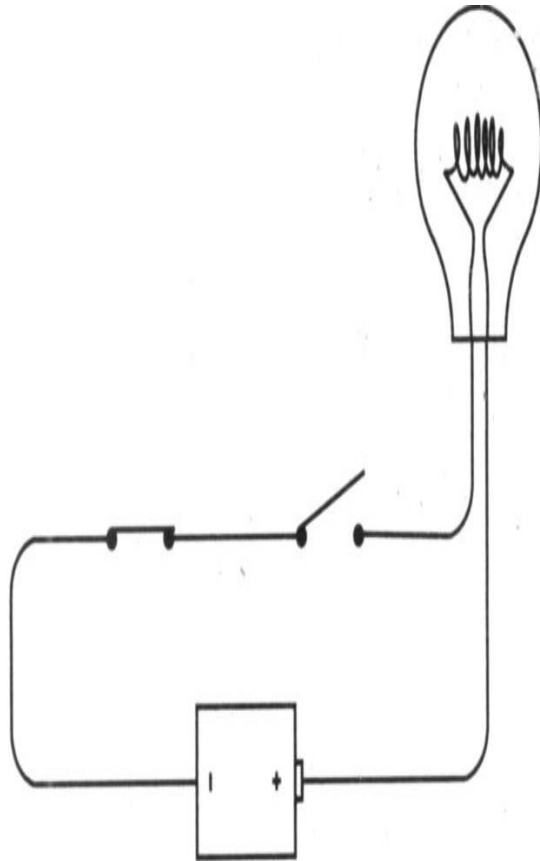
最后的结果 1 表示“是的”、“正确”，这只小猫总算找到新家了！在你买到小猫的那天晚上，当小猫蜷身睡在你的腿上时，你开始考虑是否能够通过电线

连接一些开关和灯泡来决定哪些小猫满足你的要求。（你真是一个奇怪的家伙。）你丝毫没有意识到你将要实现一个关键概念上的突破。你要做的是一些试验，这些试验把布尔代数和电路结合在一起，从而使使用二进制数字工作的计算机的设计和制造成为可能。（可别让这些话吓着你。）

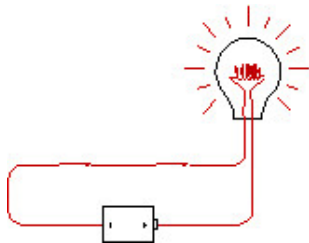
下面就开始了。你像往常一样把灯泡和电池连接在一起，这一回你用了两个开关：



开关这种方式的连接——一个在另一个的右边——称为串联的。如果你闭合了左边的开关，什么也不会发生：



同样，如果你让左边的开关断开而闭合右边的开关，结果还是一样。只有当左右两个开关都闭合时，灯泡才会发光，如下所示：



这里的关键是“都”。只有左边和右边的开关都闭合时，电流才能流过回路。这个电路执行了一个逻辑运算。事实上，灯泡回答了这个问题：“两个开关都处于闭合状

态吗？”可以把电路的工作总结为下面这张表：

左开关状态	右开关状态	灯泡状态
断开	断开	不亮
断开	闭合	不亮
闭合	断开	不亮
闭合	闭合	亮

在前一章中，我们已知道二进制数字（或“位”）是如何表示信息的：它可以表示从最普通的数字到Roger Ebert的拇指方向等的一切事情。可以说“0”代表“Ebert拇指向下的方向”，而“1”表示“Ebert拇指向上的方向”。一个开关有两个位置，所以它可以代表一个位。“0”表示“开关是断开的”，而“1”表示“开关是闭合的”。一个灯泡有两种状态，所以它也可以表示一个二进制位。“0”表示“灯泡不亮”而“1”表示“灯泡亮”。现在可以把上面的表简化一下：

左开关状态	右开关状态	灯泡状态
0	0	0
0	1	0
1	0	0
1	1	1

注意，如果交换左、右开关，结果是一样的，所以没必要指明哪个开关是左开关或右开关。因此这张表可以重画成类似于前面“AND”表和“OR”表的样子：

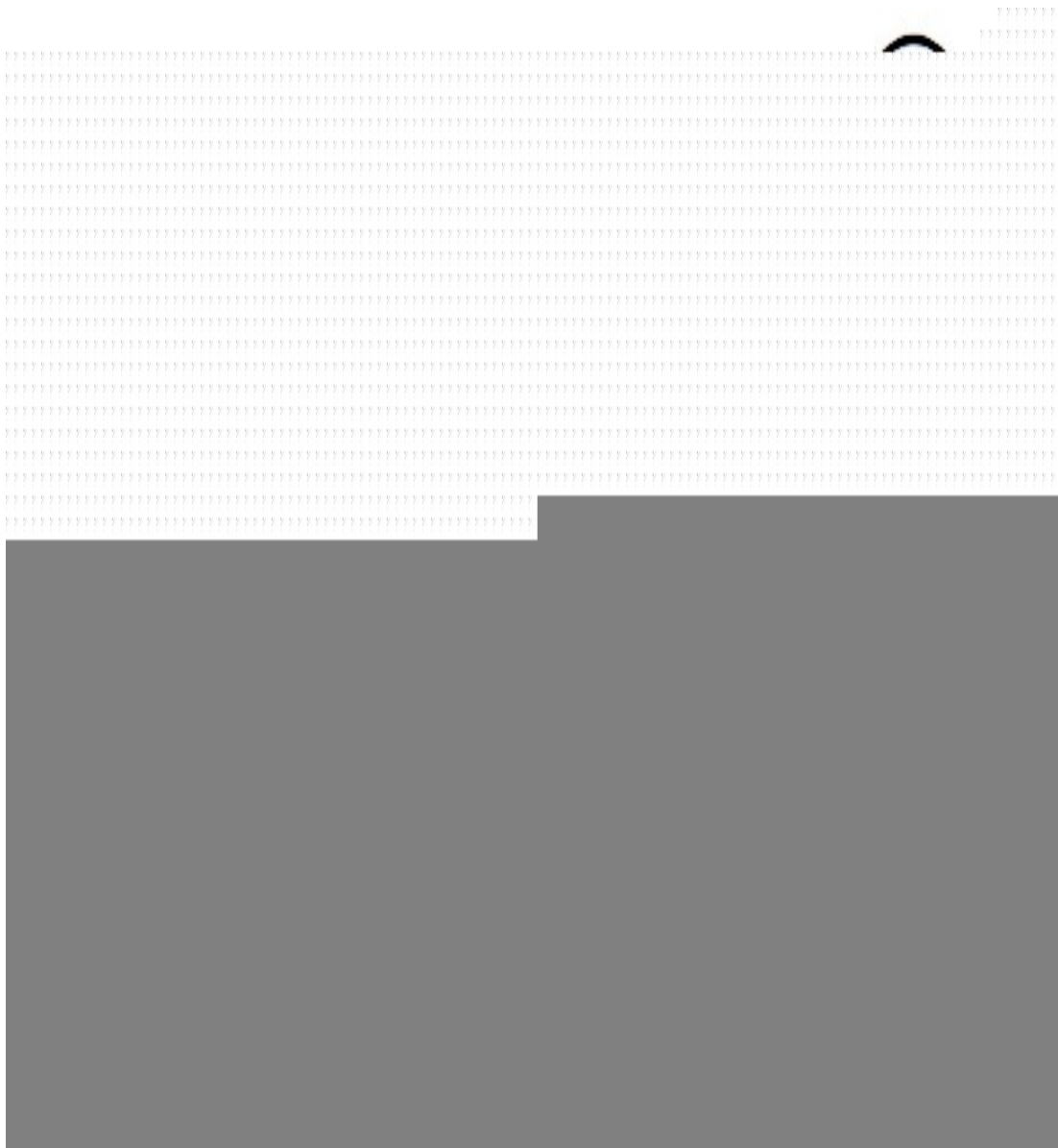
开关串联	0	1
0	0	0
1	0	1

事实上，这和“AND”表是一样的。让我们检查一下：

AND	0	1
0	0	0
1	0	1

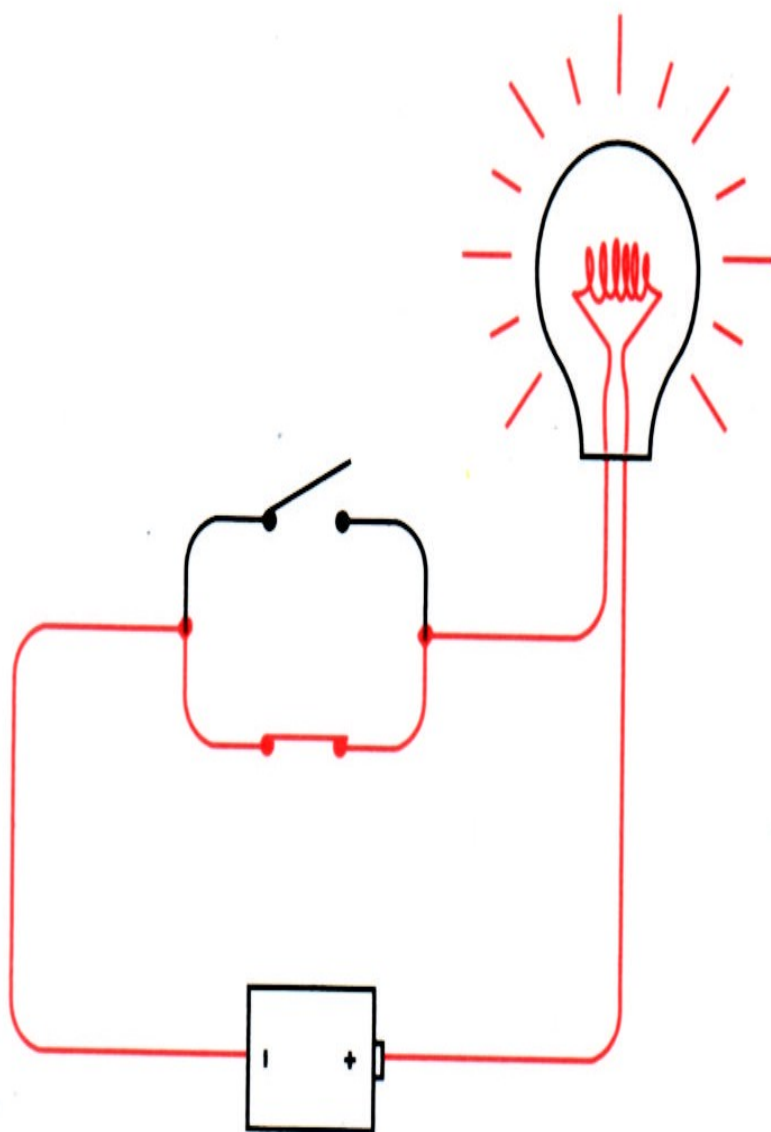
这个简单的电路实际上执行了布尔代数的“AND”操作。现在试着用另一种方式连接电路：



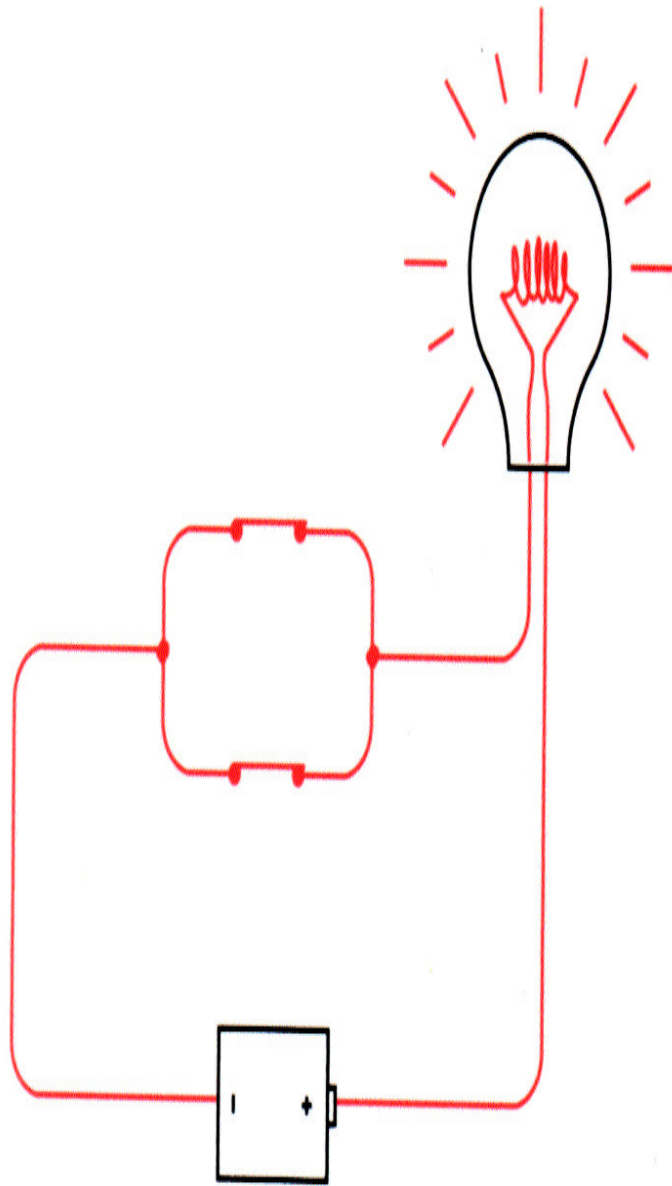


这些开关称为并行连接。它和前一种连接方式的区别是，如果闭合了上面的开关，灯泡 会亮：

如果闭合了下面的开关，灯泡会亮：



如果同时闭合上、下两个开关，灯泡还是会亮：



可见，当上面或下面的开关有一个闭合时，灯泡就会亮。这里的关键字是“或”。这个电路也执行了一个逻辑运算，灯泡回答了这样一个问题：“是否有开关闭合？”下面

的表总结了 这个电路是如何工作的:

上开关状态	下开关状态	灯泡状态
打开	打开	不亮
打开	闭合	亮
闭合	打开	亮
闭合	闭合	亮

仍然用“0”表示开关断开或灯泡不亮，用“1”表示开关闭合或灯泡亮。
这张表可以这 样:

上开关状态	下开关状态	灯泡状态
0	0	0
0	1	1
1	0	1
1	1	1

同样，这两个开关交换位置也没关系，所以这张表可以重写成如下的样子:

开关并 联	0	1
0	0	1

1

1

1

你可能已经猜到了这和布尔代数中的“OR”表是一样的：

OR

0

1

0

0

1

1

1

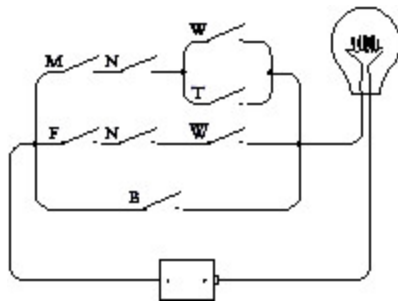
1

这意味着两个并联的开关执行的是和布尔一样的操作。当你再进入宠物店时，你告诉店员：“我想要一只阉过的公猫，白的或黄褐色的均可；或

者要一只没生育能力的母猫，除了白色，其他任何颜色均可；或者只要是只黑猫，我也要。”店员便得到了如下的表达式：

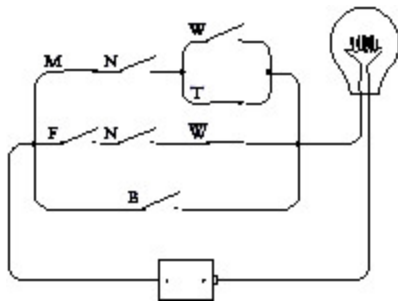
$$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$$

现在你知道两个串联开关执行的是逻辑与（AND，由符号×来表示），两个并联开关执行的是逻辑或（OR，由符号+来表示），你可以按如下方法连接8个开关：

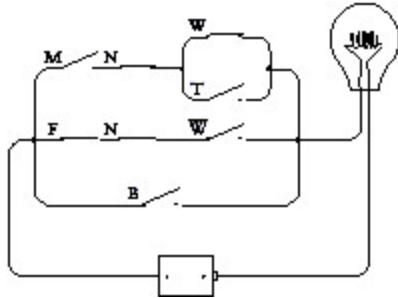


这个电路中的每一个开关都被标上了一个字母（与布尔表达式中所用字母相同）。w 表示 非W，是 $1 - W$ 的另一种写法。事实上，如果按从左至右，从上至下的顺序来阅读这个电路图，你遇到的字母的顺序和它们在布尔表达式中出现的次序是一样的。表达式中的乘号(×)都对应 角是电路图中串联的两个或两组开关的位置；表达式中的加号 (+)号对应的是电路图中并联的 两个或两组开关的位置。

你应该记得，店员最先挑出的是只未阉过的褐色的公猫。闭合相应的开关：



尽管M、T和非W这三个开关都闭合了，但没有构造出一个完整的电路来点亮灯泡。接着， 店员拿出一只无生育能力的白色的母猫：



这次，由于右边开关未闭合也无法构成一个完整的电路。但最后，店员拿出一只无生育

能力的灰色的母猫:



这样就可以构出一个完整的电路，灯泡被点亮并表示小猫符合你的要求。乔治·布尔从来没有连接过这样一个电路，他也没能看到用开关、电线和灯泡来实现一

个布尔表达式。当然，其中的一个原因是直到布尔死后 15 年，白炽灯泡才被发明。但摩尔斯在 1844 年展示了他的电报机，比布尔的《**The Laws of Thought**》的发表早 10 年，而用一个电报发声器来代替所示电路中的灯泡是十分简单的。

可惜 19 世纪没有人把布尔代数中的与、或和串联、并联一些简单的开关联系起来。数学家没有、电工没有、电报操作员也没有，没有人想到过这种联系，甚至连计算机革命的创始人查尔斯·巴贝芝（1792—1871）也没有。他曾和布尔联系过，并了解过他的工作，他一生中大部分时间致力于设计第一台差分机及接下来的解析机。一个世纪之后，这些机器被认为是现代计算机的雏型。我们现在知道，帮助巴贝芝的是他认识到计算机应产生于电报继电器中，而非那些齿轮和控制杆。

是的，问题的答案正是电报继电器。

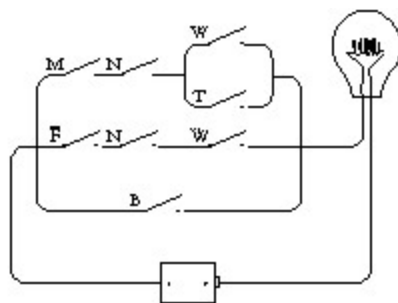
第 11 章 逻辑门电路

在遥远的将来，当人们回顾20世纪的计算机发展史时，有人可能会以为一种称为“logic gates

（逻辑门）”的设备是以著名的微软公司创始人的名字命名的（**Bill Gates**中的**Gates**在英语中有“门”的意思），其实并非如此。我们很快就会明白，逻辑门和通常让水和人通过的门十分相似。逻辑门通过阻挡或允许电流通过在逻辑中执行简单的任务。

回忆一下在上一章中你走进一个宠物店所要的那只猫，这可以由下面的布尔表达式说明：

$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$ 同时，也可以用下面的电路来选择符合条件的小猫：



这样一个电路有时被称为网络。但在今天，网络这个词更多地被用来指连接起来的计算机，而不仅仅只是开关的集合。

尽管这个电路包含的全是 19 世纪发明的东西，但那时却没有人意识到布尔代数可以直接

由电路实现。这种等同性直到 20 世纪 30 年代才被发现，主要贡献人是克劳德·香农 (生于 1916 年)。香农在他著名的、于 1938 年在麻省理工学院所写的硕士论文《A Symbolic Analysis of Relay and Switching Circuits》中阐述了这个问题。(10 年之后，香农的文章 The Mathematical Theory of Communication》是使用“位 (bit)”这个字来表示二进制数字的第 1 篇出版物。)

1938 年以前，人们已经知道当把两个开关串联起来时，只有两个开关都闭合电流才能流通；而当把两个开关并联起来时，只需闭合其中的一个即可构成回路。但没有人能像香农那样清晰地阐述电子工程师可以使用布尔代数的所有工具来设计带开关的电路。此外，如果你简化了描述网络的布尔表达式，你也可以相应地简化网络。

例如，描述你想要的小猫的表达式是：

$(M \times N \times (W + T)) + (F \times N \times (1 - W)) + B$ 用结合律把用 \times 结合的变量重新排序并按下面的方式重写表达式：

$$(N \times M \times (W + T)) + (N \times F \times (1 - W)) + B$$

为更清楚地表达意图，可以定义名为 X 和 Y 的两个新变量：

$$X = M \times (W + T)$$

$$Y = F \times (1 - W)$$

现在，描述你想要的小猫的表达式可以写成下面的样子：

$(N \times X) + (N \times Y) + B$ 完成简化后，我们再把 X 、 Y 代回原来的式子。

注意，变量 N 在表达式中出现了两次。使用分配律，表达式可以按如下方式重写，并只使

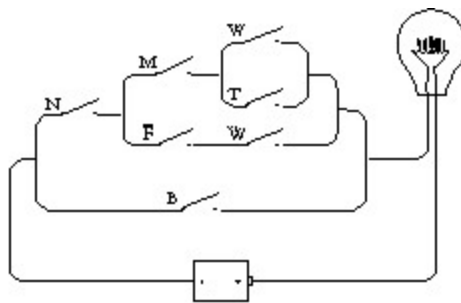
用一个 N ：

现在把 \mathbf{X} 、 \mathbf{Y} 表达式代入：

$$(N \times (X+Y)) + B$$

$(N \times ((M \times (W+T)) + (F \times (1-W)))) + B$ 由于有很多圆括号，该表达式看上去似乎仍很复杂。但表达式中少了一个变量项（减少

了一次 \times 运算），也就意味着网络中少了一个开关。这是修改后的电路图：



确实，证明修改前后的两个电路图功能是一样的比去证明两个表达式功能是相同的要简单。

可是，网络中仍然多余了三个开关。理论上讲，你只需要四个开关来定义你心目中的猫咪。为什么是四个呢？因为每个开关都是一个“位”。你需要一个开关来定义性别（断开表示公的，而闭合表示母的）；一个开关来定义是否有生育能力（闭合表示阉过的，断开表示未阉过的）还需要两个开关表示颜色。因为只有四种可能的颜色（白、黑、褐和其他所有颜色），而我们知道四种选择可以用两个二进制位来定义，所以只需要两个开关来表示颜色。例如，两个开关都断开表示白色，一个闭合表示黑色，另一个闭合表示褐色，两个开关都闭合就表示其他所有颜色。

现在，让我们做一个控制面板来选择一只猫。控制面板上有四个开关（正如你家里的电灯开关）和一个灯泡：



控制面板

开关打到上面是指开关闭合，反之是指开关断开。也许表示猫的颜色两个开关标识得不是很清楚，这是为了把控制面板做得更简练不得已而造成的。在表示颜色的一对开关中，左边的开关标着**B**，如果只有它往上就表示黑色；右边的开关标着**T**，如果只有它往上就表示黄褐色；**B**、**T**两个开关均往上则表示其他颜色，由**O**标识；**B**、**T**两个开关均往下则表示白色，由**W**标识。

在计算机专业术语中，开关是一种输入设备，输入是控制电路如何工作的信息。本例中输入开关对应于描述一只猫咪的4位信息，输出设备是灯泡。如果开关描述了一只符合条件的猫，灯泡就会亮。上面介绍的控制面板上的开关被设置成表示一只无生育能力的黑母猫，这是符合你的要求的，所以灯泡亮了。

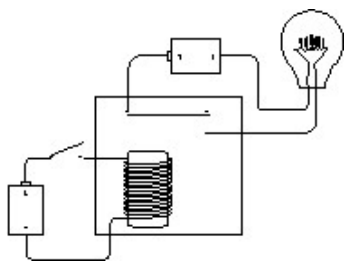
现在所要做的是设计一个使控制面板工作的电路。

前面提到过香农的论文题目是《**A Symbolic Analysis of Relay and Switching Circuits**》，他所指的 **relay** 和第6章中所讲的电报系统的继电器很类似。在香农的论文发表时，继电器已被用作其他目的，尤其是用于电话系统的大型网络。

像开关一样，继电器也可以串联或并联以执行逻辑中的简单任务。继电器的组合称为逻辑门。这里所说的“逻辑门执行简单逻辑任务”是指逻辑门只完成最基本的功能。继电器比开关好是因为继电器可以被其他继电器控制而不必用手指控制，这意味着逻辑门可以被组合起来以执行更复杂的任务，比如一些简单的算术操作。事实上，下一章就要展示如何用电线连接开关、灯泡、电池和继电器来构造一个加法机（尽管它只能工作于二进制数字状态）。

继电器对电报系统的工作十分重要。连接电报站的电线长距离时电阻很大，需要一种方法来接收微弱的信号并把它增强后发送出去。继电器通过使用电磁铁控制开关可做到这一点。事实上，继电器放大了一个很弱的信号使其成为一个强信号。

就我们的目的而言，我们并不对它的信号放大能力感兴趣，真正使我们着迷的是继电器作为开关可用电来控制而不用手指。可以用电线把继电器、开关、灯泡和一对电池做如下连接：



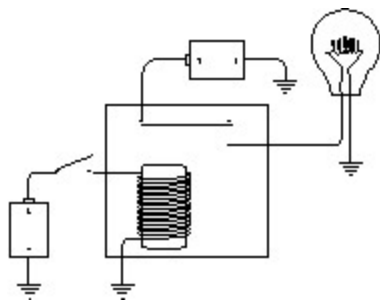
注意左边的开关是断开的，灯泡不亮。当闭合开关时，电流流过围绕在铁棒上的线圈，于是铁棒具有了磁性，并把上面有弹性的金属簧片拉下来，从而连通了电路，使灯泡发光：



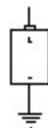
当电磁铁把上面的金属簧片拉下来时，这个继电器被称为“触发了”。当左边的开关断开时，铁棒不再有磁性，继电器中的金属簧片则弹回到原来的位置。

这看上去似乎是用一种很不直接的方式点亮灯泡的，但实际上这种方式是很直接的。如果我们只对点亮灯泡感兴趣，我们完全可以舍弃继电器。但我们的兴趣并非只是点亮灯泡这么简单，我们有更宏伟的目标。

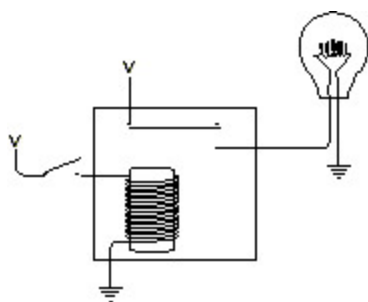
本章要多次用到继电器（当逻辑门建好之后就会很少再用了），所以要把上面那幅图简化一下。可以通过大地省去一些导线。在这种情况下，大地仅代表了一个公共端，并不是指真正的物理接地：



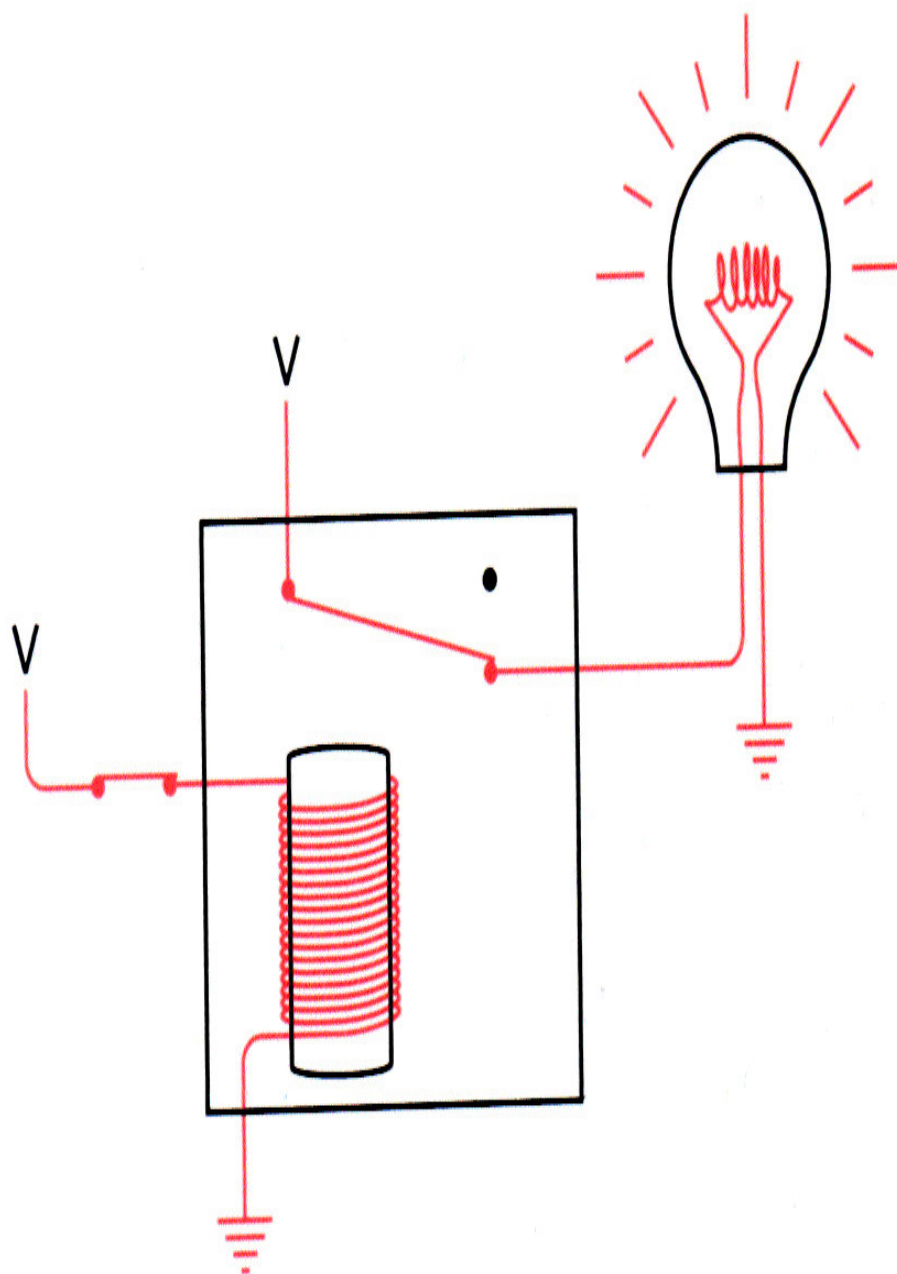
这看上去仍然不够简化，但还不至于那样做。注意两个电池的负极均接地，所以当看到的电池是这样的时：



可用大写字母“V（它代表电压）”代替上图中的电池（如在第5和第6章中所做的）。现在继电器看上去如下图所示：

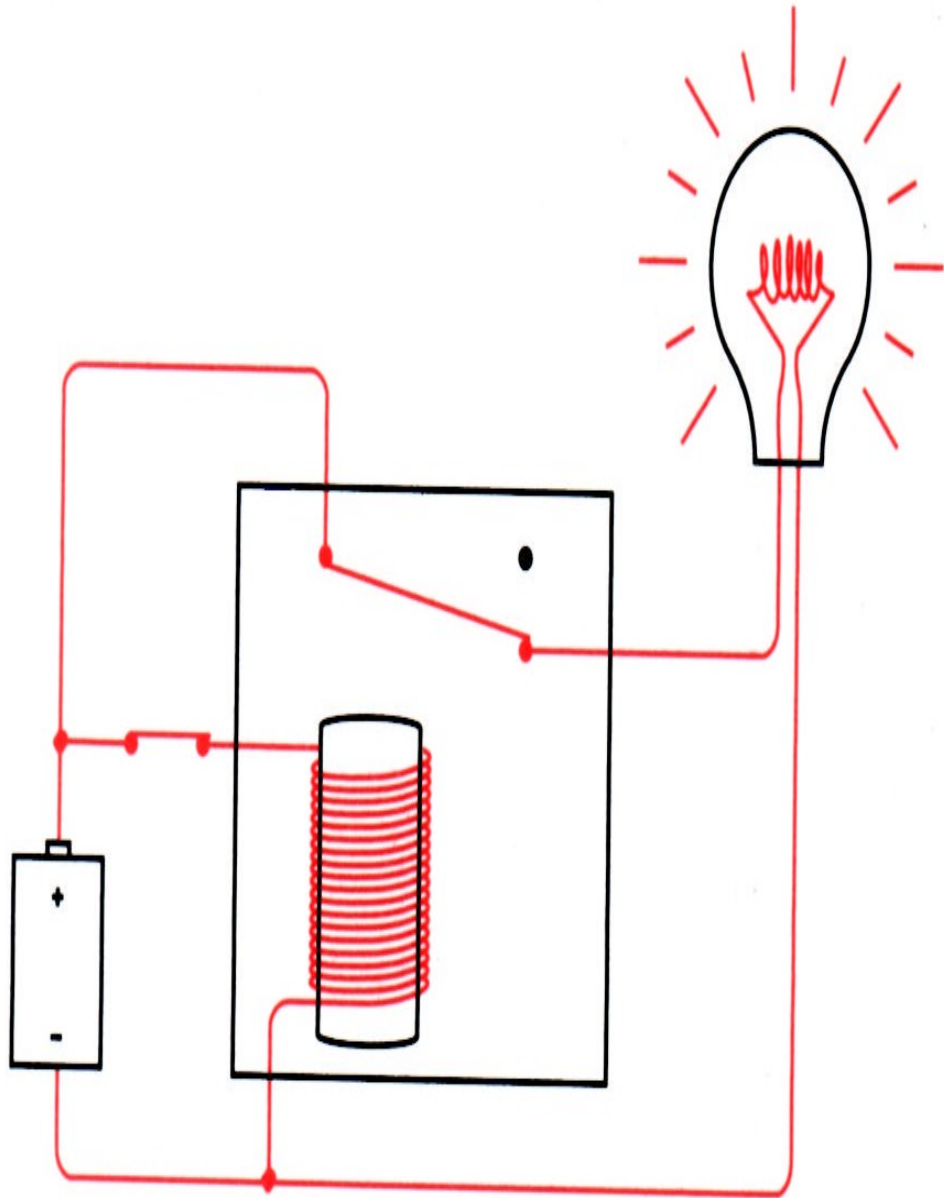


当右边开关闭合时，电流从 V 端流出，经过电磁铁芯流到地上。这使得电磁铁把上面有弹性的金属簧片拉下来，从而连通了接有灯泡的电路，灯泡点亮：

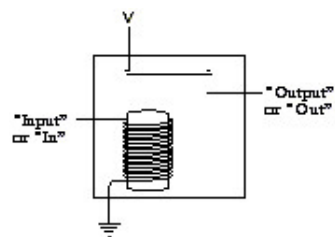


上面图显示了两个电源和两个接地，但本章的所有图中，电源，即“V”，可以互连，接

地端也可以互连。本章及下一章的所有继电器和逻辑门的网络只要求有一个电池，但可能是一个大容量的电池。例如，上一幅图可只用一个电池，如下所示：



但这幅图并不能清楚地表明要用继电器做什么。先不考虑电路而把注意力放到输入和输出上，就像前面的控制面板一样：



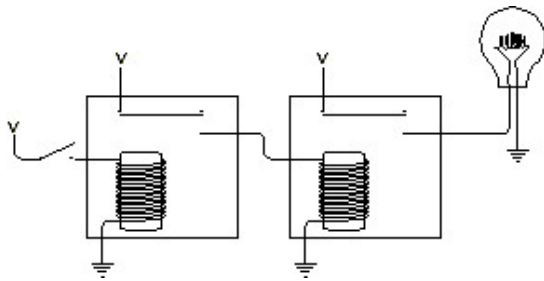
输出

输入

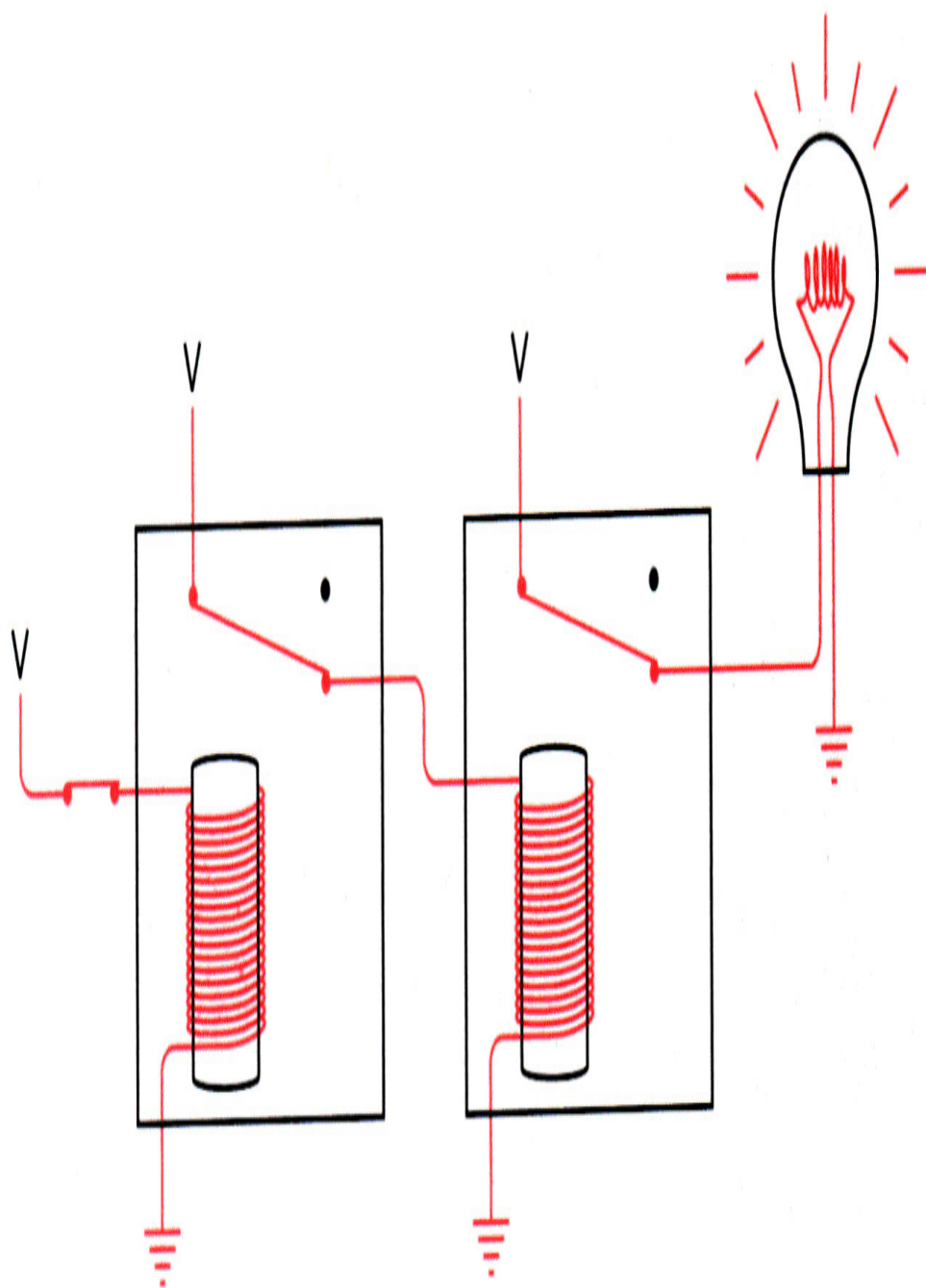
如果电流流经输入（例如，用一个开关把输入连到“V”端），电磁铁就会被触发，输出就有了一个电压。

继电器的输入不一定只能是开关，其输出也未必只限于灯泡。一个继电器的输出可以连

到另一个继电器的输入，如下所示：



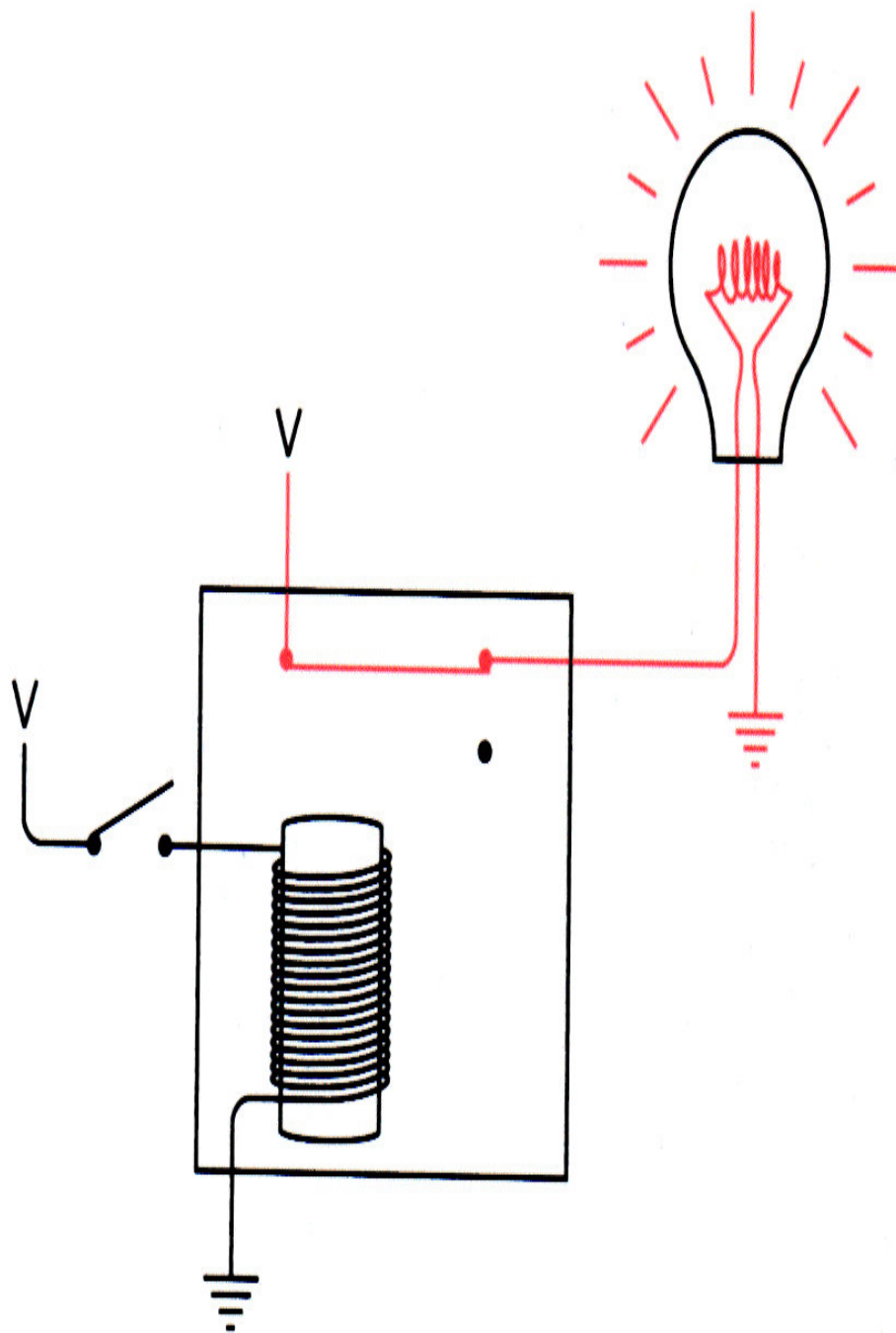
当闭合开关时，第一个继电器被触发，它为第二个继电器提供了输入电压，于是第二个继电器也被触发，灯泡被点亮了：



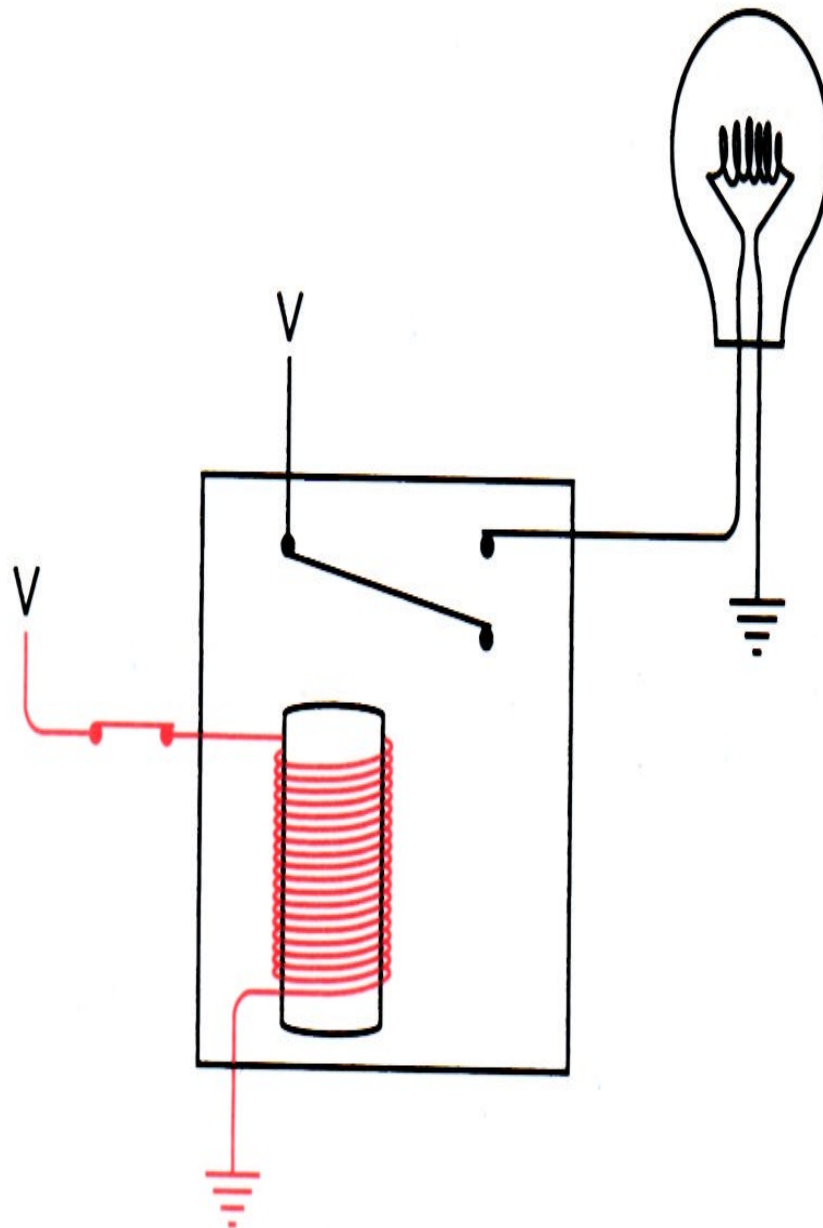
把继电器连接起来是构造逻辑门的关键。事实上，灯泡可以两种方式连到继电器上。注意，具有弹性的金属簧片是被电磁铁拉下

来的。平时，金属簧片与上端接触，当电磁铁吸引它的时候，它便和下端接触。我们一直把金属簧片与下端的接触作为继电器的输出，但

我们也可以把它与上端的接触作为输出。当使用这种输出时，结果正好相反，输入开关断开时灯泡是亮的：



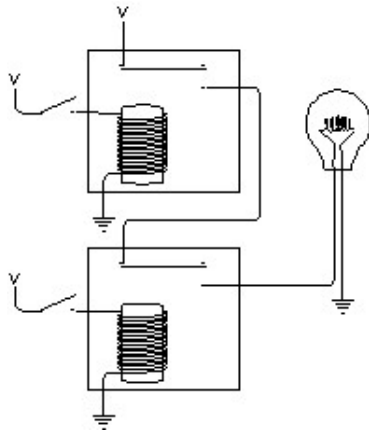
而当输入开关闭合时，灯泡便灭了：



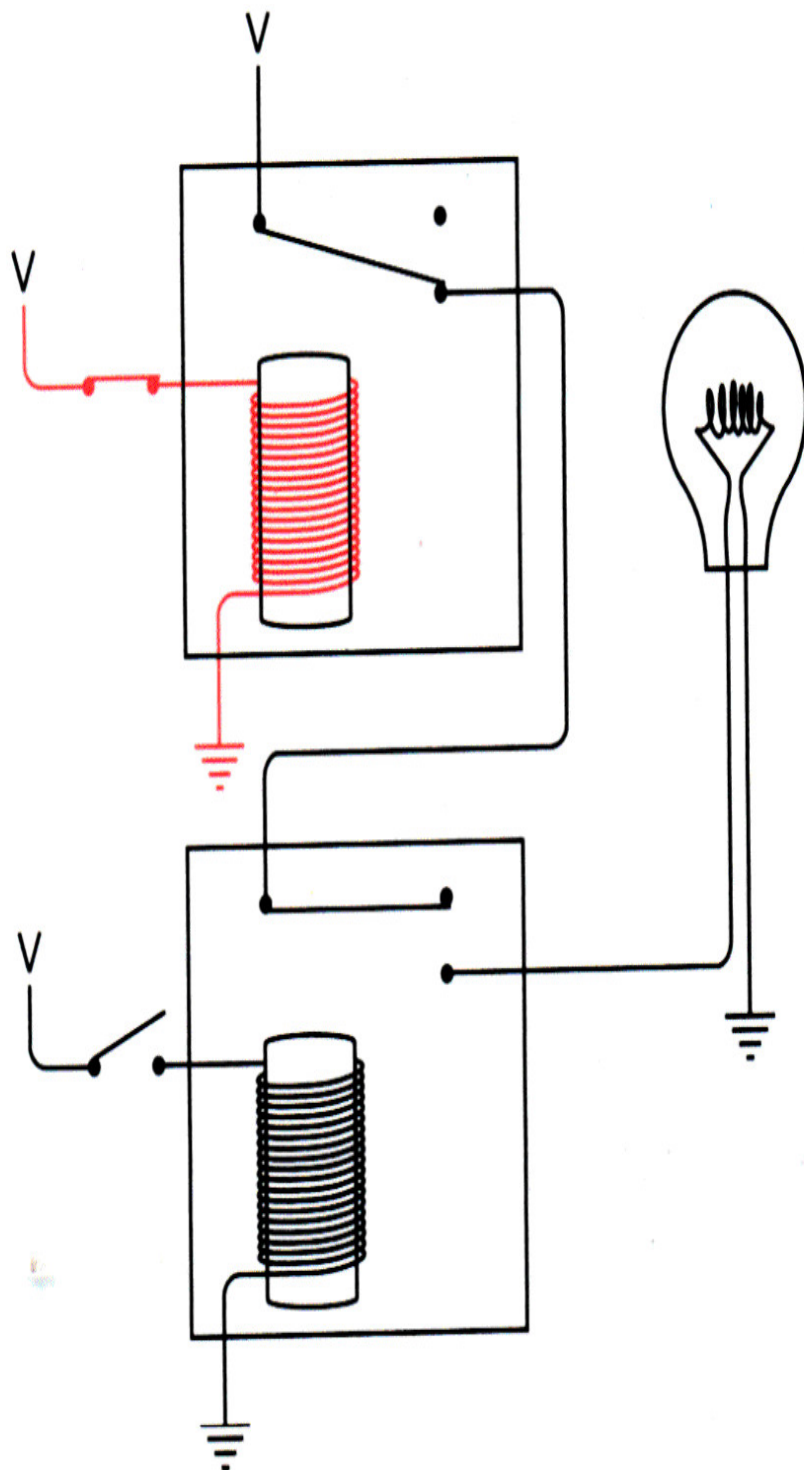
使用这种开关的继电器称为双掷继电器，它的两个输出在电性上是相反的——当一个有电压时，另一个则没有。

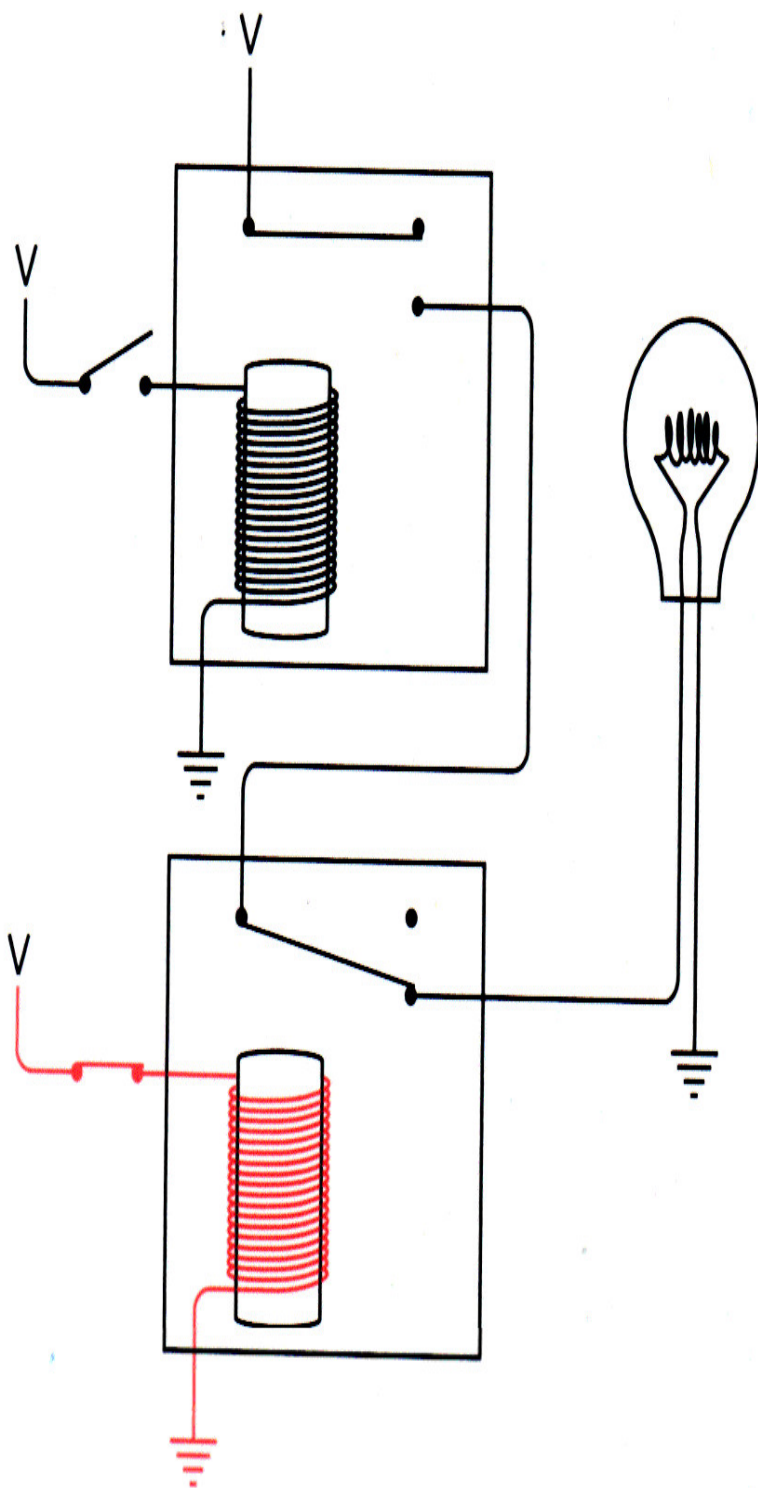
顺便说一下，如果你不知道现在的继电器是什么样子，你可以很方便地从当地的电器行的透明小包里看到一些。有些继电器就像 (加入饮料的) 方形小冰块一样大小，如元件号为 275-206 和 275-214 的继电器就是这种大小的且经久耐用的继电器。它们被封在一个干净的塑料外壳里，所以你可以看到电磁铁和弹性金属簧片。本章和下一章所描述的电路都使用的是元件号为 275-240 的继电器，它体积小且价格便宜（每个 \$ 2.99）。

正如两个开关可被串联一样，两个继电器也可以串联：



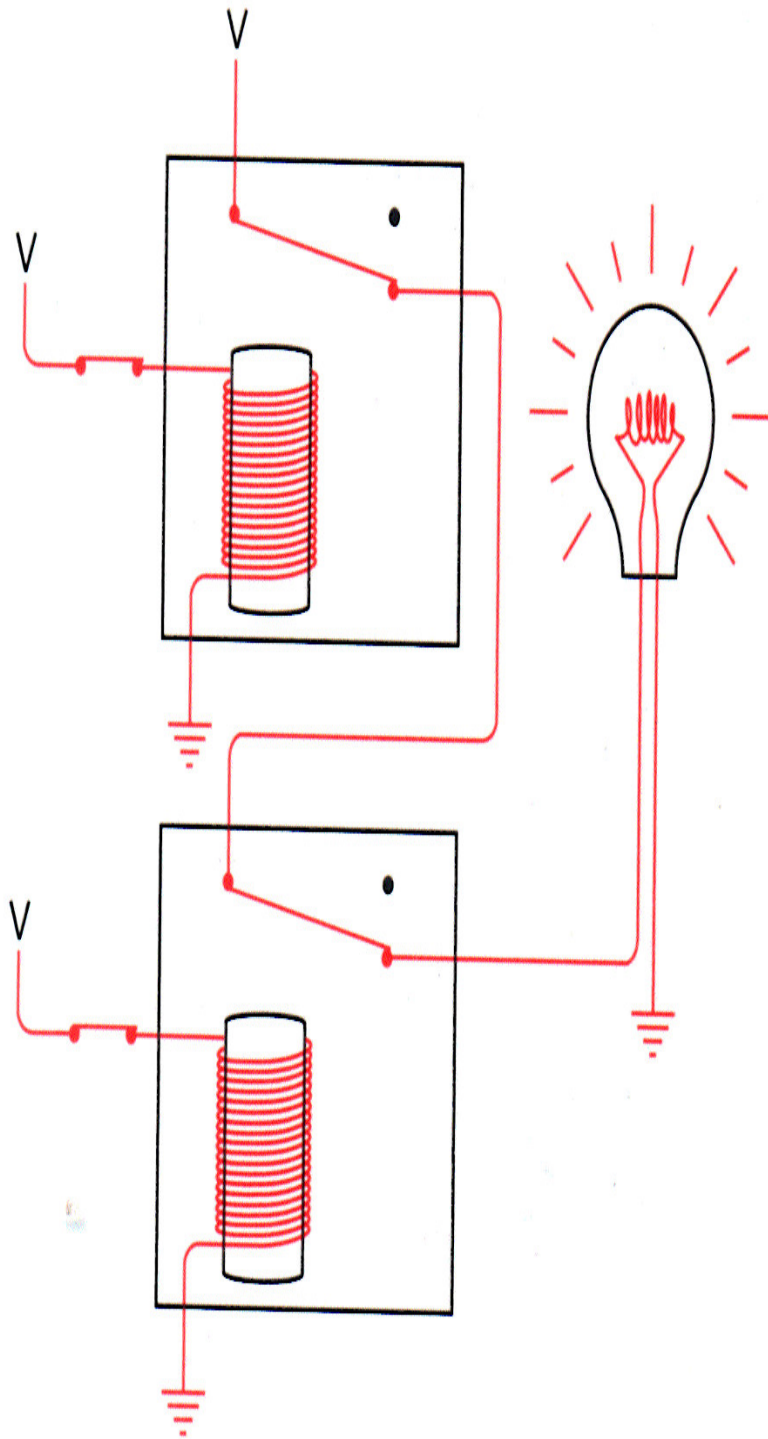
上面继电器的输出为下面的继电器提供了输入电压。如上所示，当两个开关均断开时，灯泡不会发光。试着闭合上面的开关：





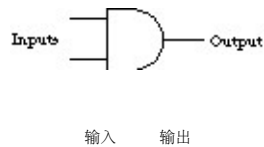
由于下面的开关仍旧断开，下面的继电器没有触发，所以灯泡仍然不亮。
若断开上面的 开关而闭合下面的开关：

灯泡仍旧不亮。因为上面的继电器未被触发，电流无法流经灯泡。点亮灯泡的唯一方法是闭合两个开关：



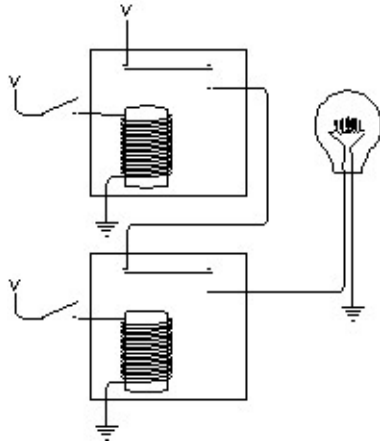
现在，两个继电器都被触发了，电流可以在电源、灯泡和接地点之间流通。同串联开关一样，这两个继电器也执行了逻辑操作。只有当两个继电器都被触发时，灯

泡才会点亮。串联的两个继电器就是一个“**AND gate**（与门）”。为避免复杂的图示，电气工程师使用一个特殊的符号表示“与门”，如下图所示：

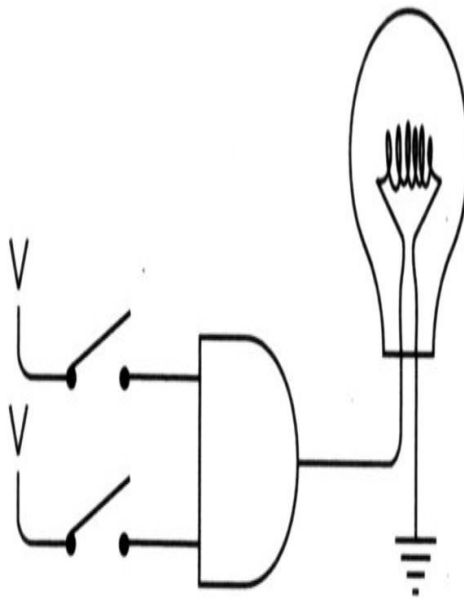


这是四个基本逻辑门中的第一个。与门有两个输入端（在图的左部），一个输出端（在图的右部）。这样表示的与门通常输入在左部，输出在右部。这是因为人们习惯于从左到右读图。但是与门同样可以画成输入在上部、右部或底下。

有两个继电器、两个开关和一个灯泡的原始电路图如下所示：

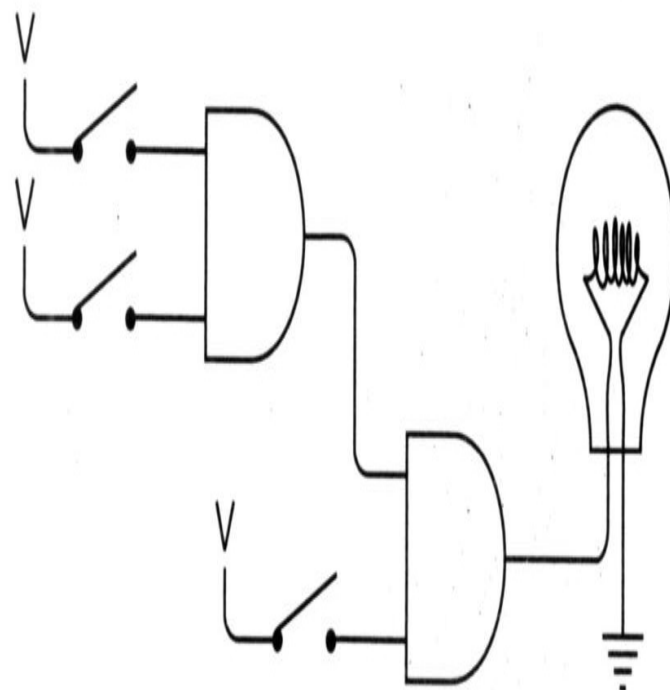


使用“与门”符号，上图可同样表示成：

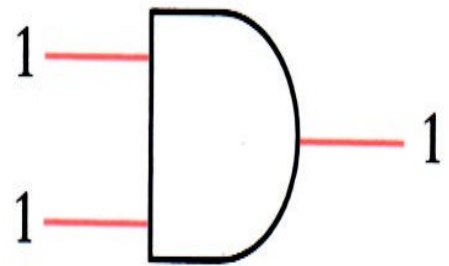
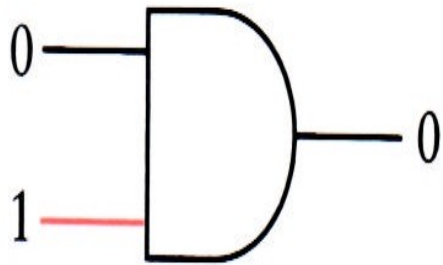
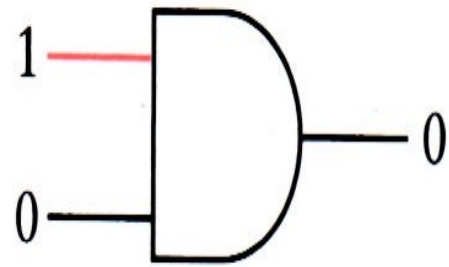
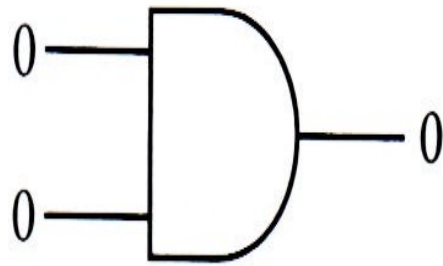


注意与门不仅代替了串联的两个继电器，同时也隐含了上面的继电器连着电源，且两个继电器都是接地的。只有当上下两个开关都闭合时，灯泡才会发光，这就是它之所以叫与门的原因。

与门的输入未必一定要和开关连接，且输出也不一定只能是灯泡。我们真正要处理的是输入端的电压和输出端的电压。例如，一个与门的输出可以是另一个与门的输入：



只有当三个开关都闭合时，灯泡才会发光。当上面的两个开关闭合时，第一个与门的输

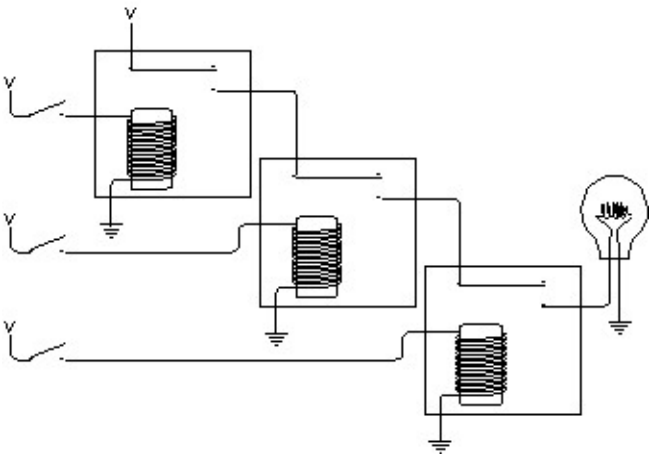


出会触发第二个与门的第一个继电器，而最下面的开关会触发第二个与门的第二个继电器。如果把不加电压视为 0，加上电压视为 1，则与门的输出按如下方式由输入来决定：

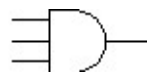
正如两个串联的开关一样，与门的输入输出关系可作如下描述：

AND	0	1
0	0	0
1	0	1

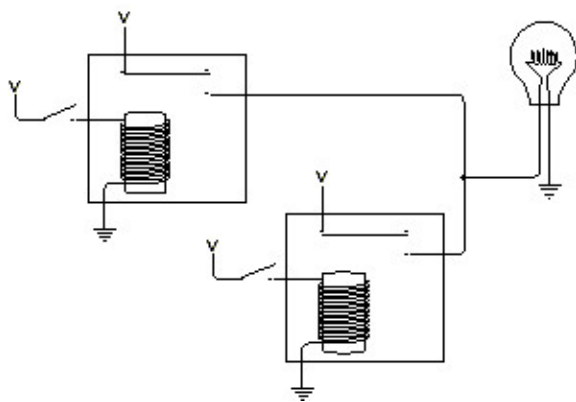
与门也可以有多于两个的输入端。例如，假设串联了三个继电器：

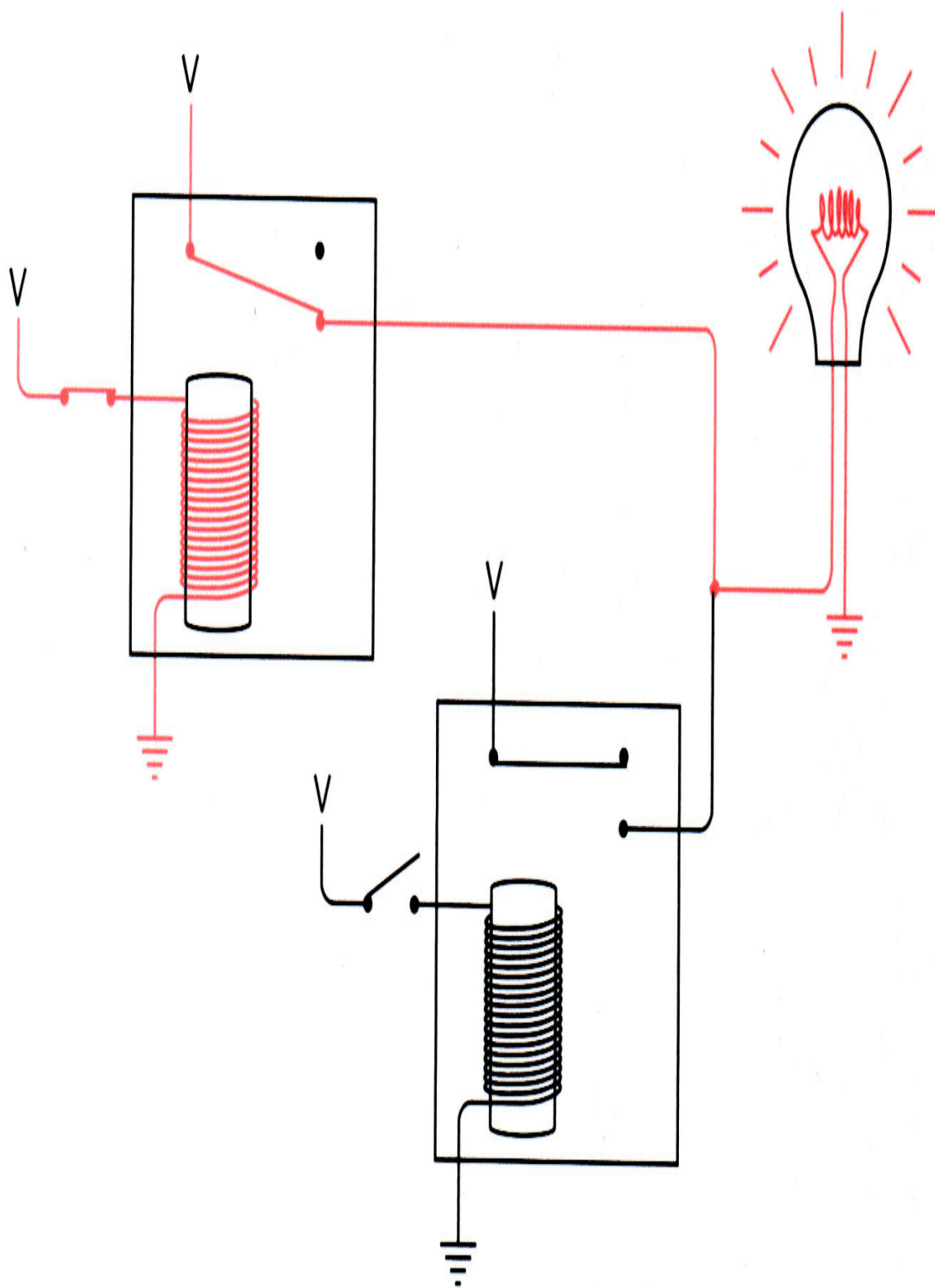


只有当三个开关同时闭合时，灯泡才会发光。这种配置可用如下符号表示：



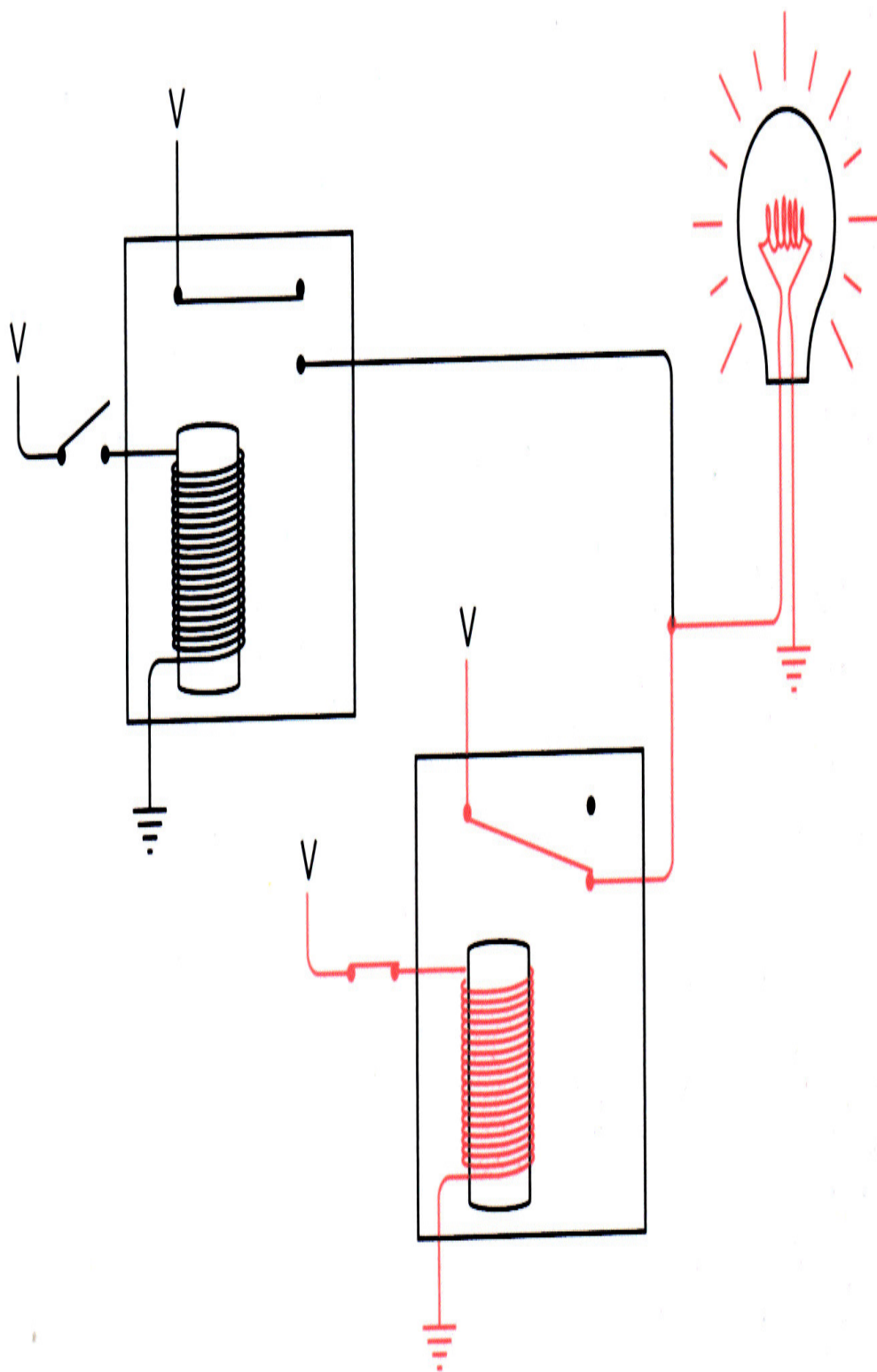
它被称为三输入端与门。以下逻辑门可用并联的继电器解释：



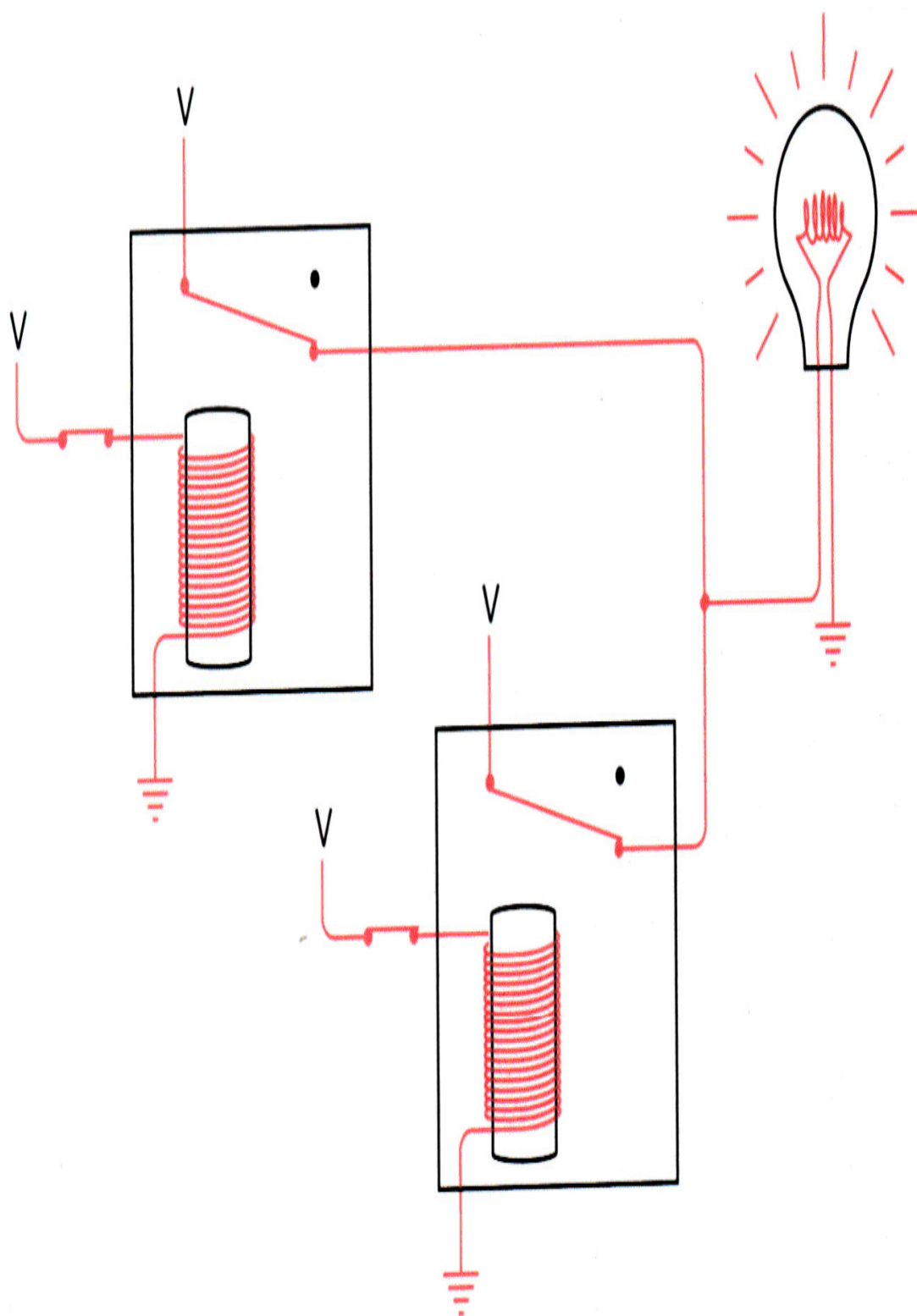


注意两个继电器的输出是连接在一起的，这个连接在一起的输出为灯泡提供了电源。任何一个继电器都可以点亮灯泡，例如，如果闭合上面的开关，灯泡会亮。这时，灯泡从左上角的继电器得到了电力供应。

如果让上面的开关断开而闭合下面的开关，灯泡也会亮：



当两个开关都闭合时，灯泡同样会亮：



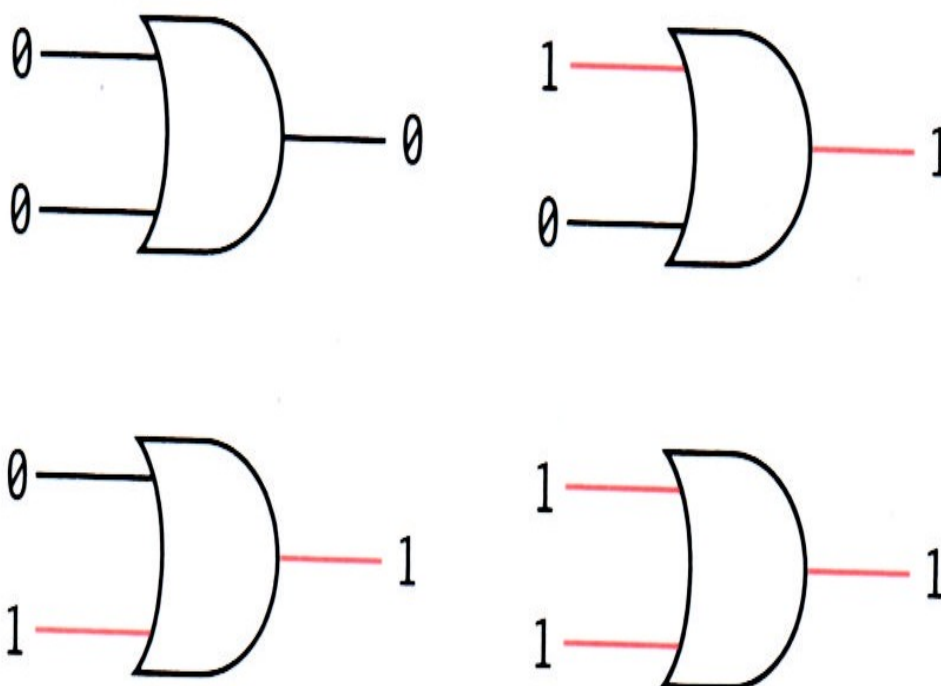
可见，当上开关或下开关中的任何一个闭合时，灯泡都会亮。这里的关键是“或”，所以这样的门叫“**OR gate**（或门）”。电气工程师使用如下符号表示或门：



输入 输出

它看上去和与门很相似，只是接输入端的一边是弧形的，很像英语“**OR**”中的字母“**O**”。或门的两个输入中，只要有一个加上电压，输出就是高电位。同样，如果约定不加电压

是0，而加电压是 1，则或门也有四种可能的组合状态：

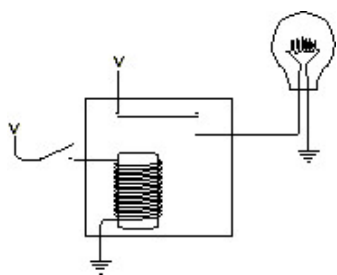


可以把或门的输入输出关系小结成如下表格：

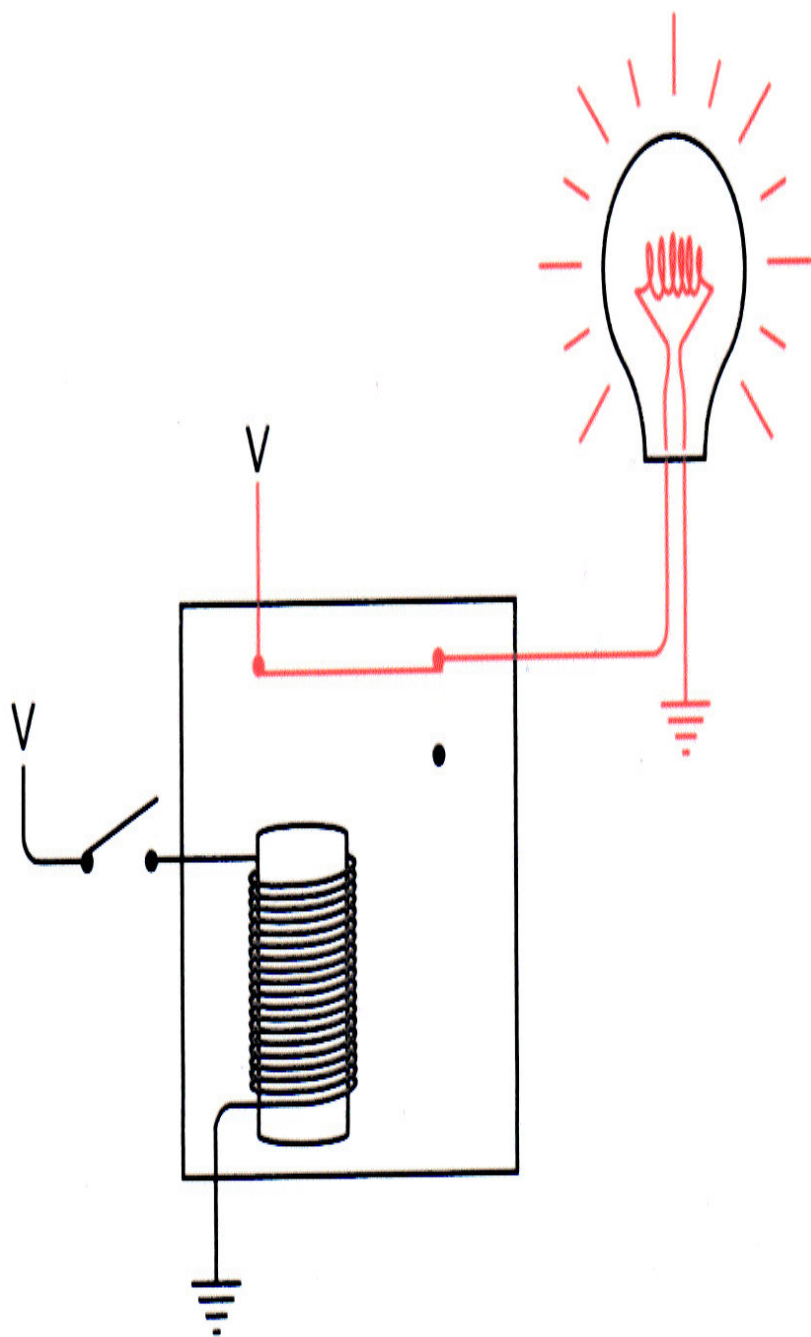
OR		0	1
0	0	0	1
1	1	1	1

或门也可以有两个以上的输入端（当任一输入端为 1 时，输出端就为 1；只有所有输入端 均为 0 时，输出端才为 0）。

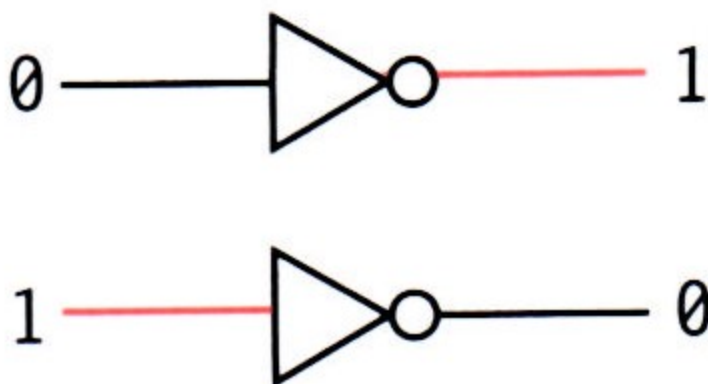
前面解释过继电器可称为双掷继电器，因为其输出可以两种不同的方式连接。通常情况下，当开关断开时，灯泡不亮：



当开关闭合时，灯泡点亮。也可以用另外一种连接方式，使开关断开时灯泡点亮：

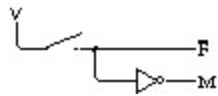


在这种情况下，只有闭合开关时灯泡才熄灭。以这种方式连接的继电器叫作 反向器 。反 向器不是逻辑门（逻辑门通常有两个以上的输入），但它十分有用。反向器可以用下面的符号 表示：

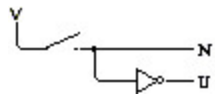


它被称为反向器的原因是当输入为 0 时输出却为 1，反之亦然：

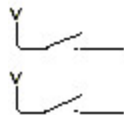
有了反向器、与门和或门，我们就可以制作控制板来自动选择理想的小猫了。让我们从开关开始。第一个开关的闭合表示母猫，断开表示公猫。这样，可以产生称为 **F** 和 **M** 的两个信号，如下图所示：



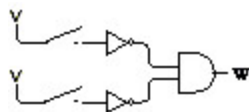
当 **F** 是 1，**M** 就是 0，反之亦然。同样，第二个开关的闭合表示阉过的猫，而断开表示有生育能力的猫：



接下来的两个开关更复杂一些，不同的组合要代表四种不同的颜色。这里有两个开关，都与电源相连：

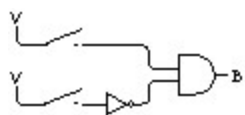


当两个开关都断开时，它们表示白色。我们用两个反向器和一个与门来产生信号 **W**。如果选择了一只白猫，**W** 就为 1，否则为 0：

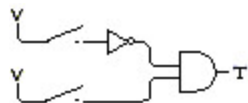


当开关断开时，两个反向器的输入是 0，这样反向器的输出（也就是与门的输入）为 1，这也就意味着与门的输出为 1。一旦一个开关闭合，与门输出即为 0。

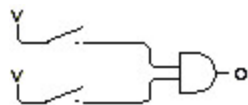
为表示一只黑猫，闭合第一个开关，这可以用一个反向器和一个与门实现：



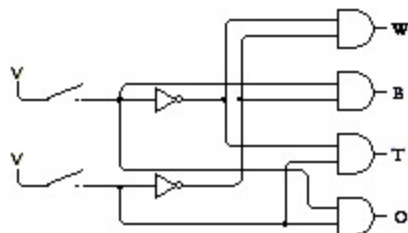
只有当第一个开关闭合而第二个开关断开时，与门的输出才是 1。同样，当第二个开关闭合而第一个开关断开时，与门的输出也为 1。我们用来表示褐色：



而如果两个开关都闭合时，用如下图示表示其他颜色：



现在把四个小电路集成为一个大电路（通常，黑点表示电线的连接点，没有黑点的交叉线是不连接的）：



这个连接图看起来十分复杂。但如果仔细地沿着线路走，看清楚每个与门的输入而不要关心这些输入又连到了别的什么地方，你就会明白电路是如何工作的。如果两个开关都断开，信号 **W** 会是 1，其余信号都是 0。如果第一个开关闭合，则信号 **B** 会是 1，其余信号都是 0。

连接门和反向器时可以遵循一些简单的规则：一个门（或反向器）的输出可以作为其他门（或反向器）的输入，但是两个以上的门（或反向器）的输出永远不能互连在一起。

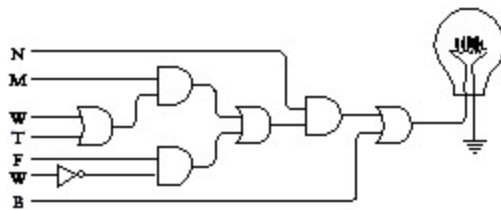
由 4 个与门和 2 个反向器组成的电路叫作“2-4 译码器”。输入是两个二进制位的不同组合，共代表了 4 个不同的值。输出是 4 个信

号，任何时刻只能有一个是 1，至于哪一个是 1 取决于两个输入位。用同样的原理还可以构造“3-8 译码器”或“4-16 译码器”等等。

选择小猫的表达式的简化表示是：

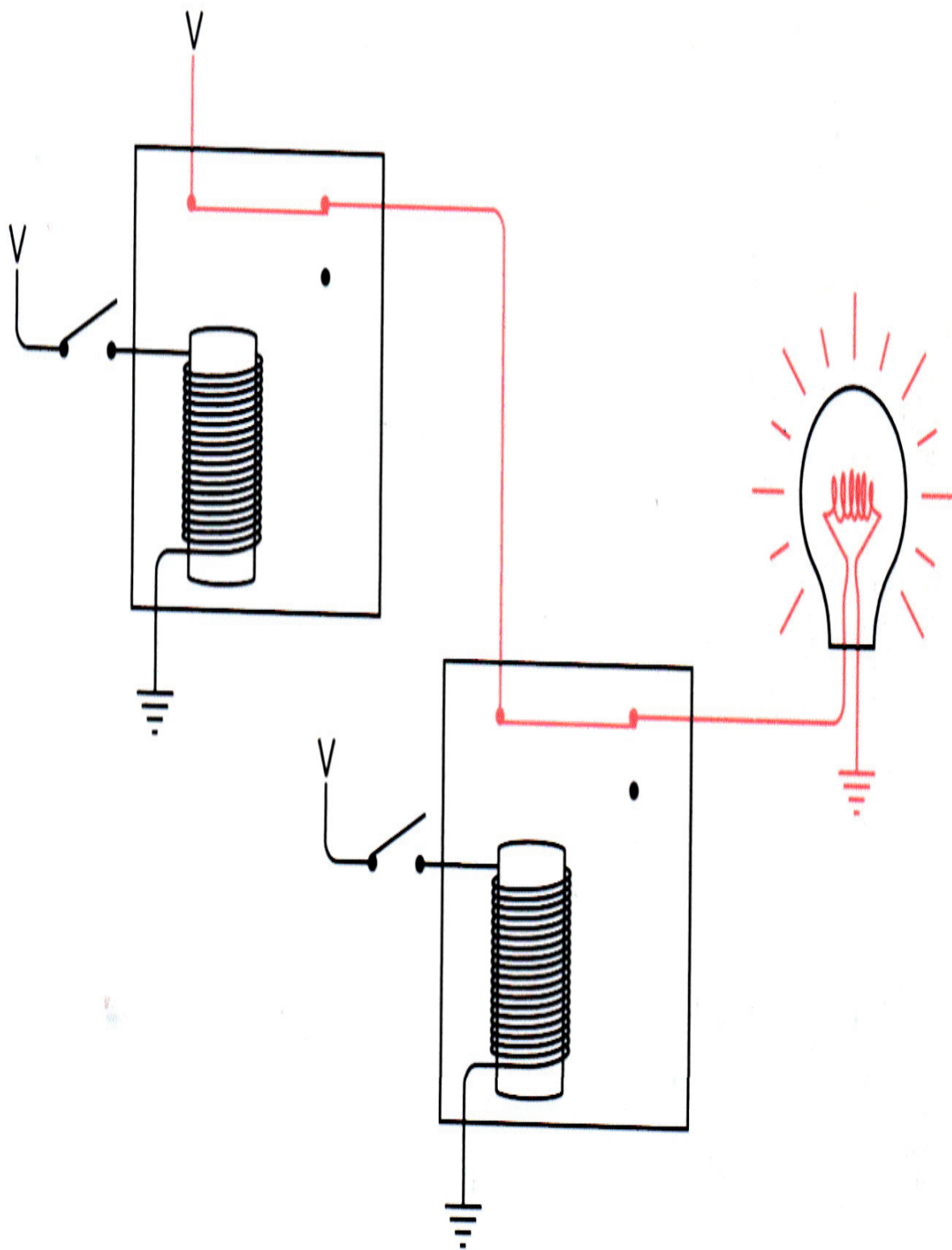
$(N \times ((M \times (W + T)) + (F \times (1 - W)))) + B$ 对于表达式中的每一个加号 (+)，必定对应电路中的一个或门。对于每一个乘号 (\times)，则

对应一个与门：



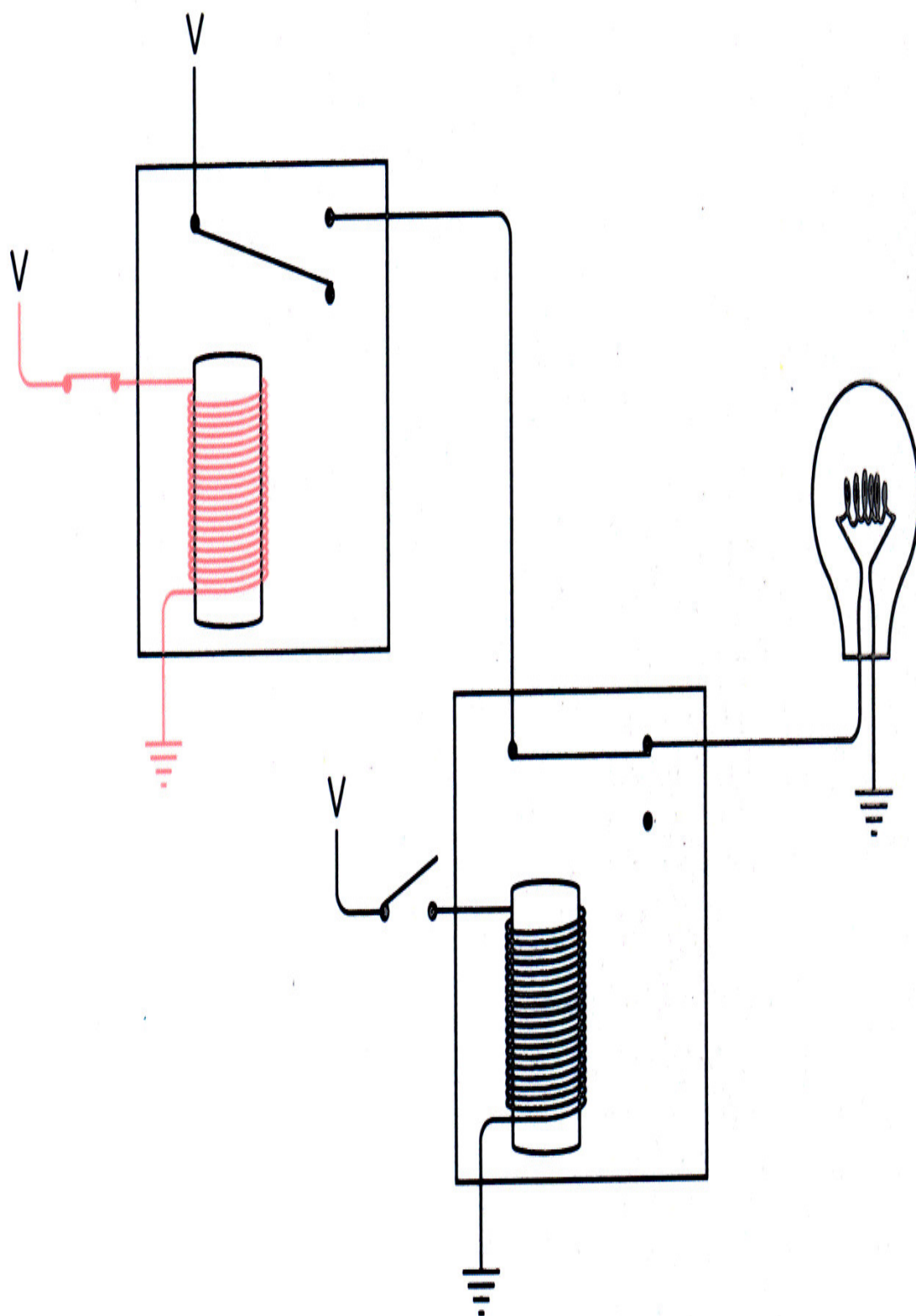
电路图左边的符号和它们在表达式中出现的顺序是一样的。这些信号来自于和反向器连接的开关及 2-4 译码器的输出。注意，图中用了反向器来表示表达式中的 $(1-W)$ 。

你可能会说：“这不过是一堆继电器而已。”不错，这正是一堆继电器，每个与门和或门中都有两个继电器，一个反向器中有一个继电器，因而只能说你必须习惯它。以后的各章会用更多的继电器。不过，所幸的是你不用真正地去买一堆回家连起来。

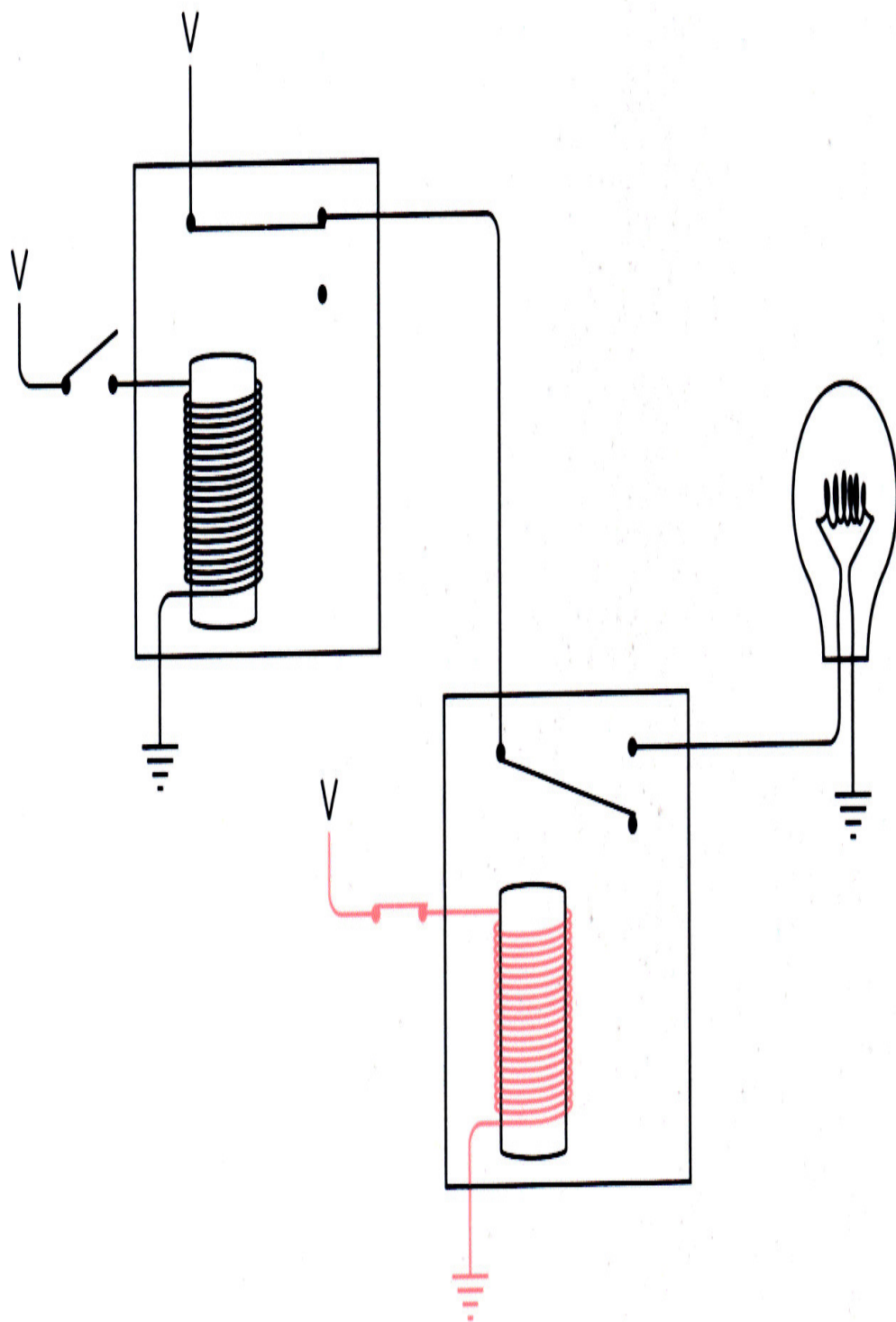


本章再看两个逻辑门。这两个门都会用到这样一个继电器，该继电器在不被触发时，其输出为高电位（这是用在反向器中的输出）。例如，下面配置中，一个继电器的输出为第二个继电器提供了电源。当两个输入都断开时，灯泡是点亮的：

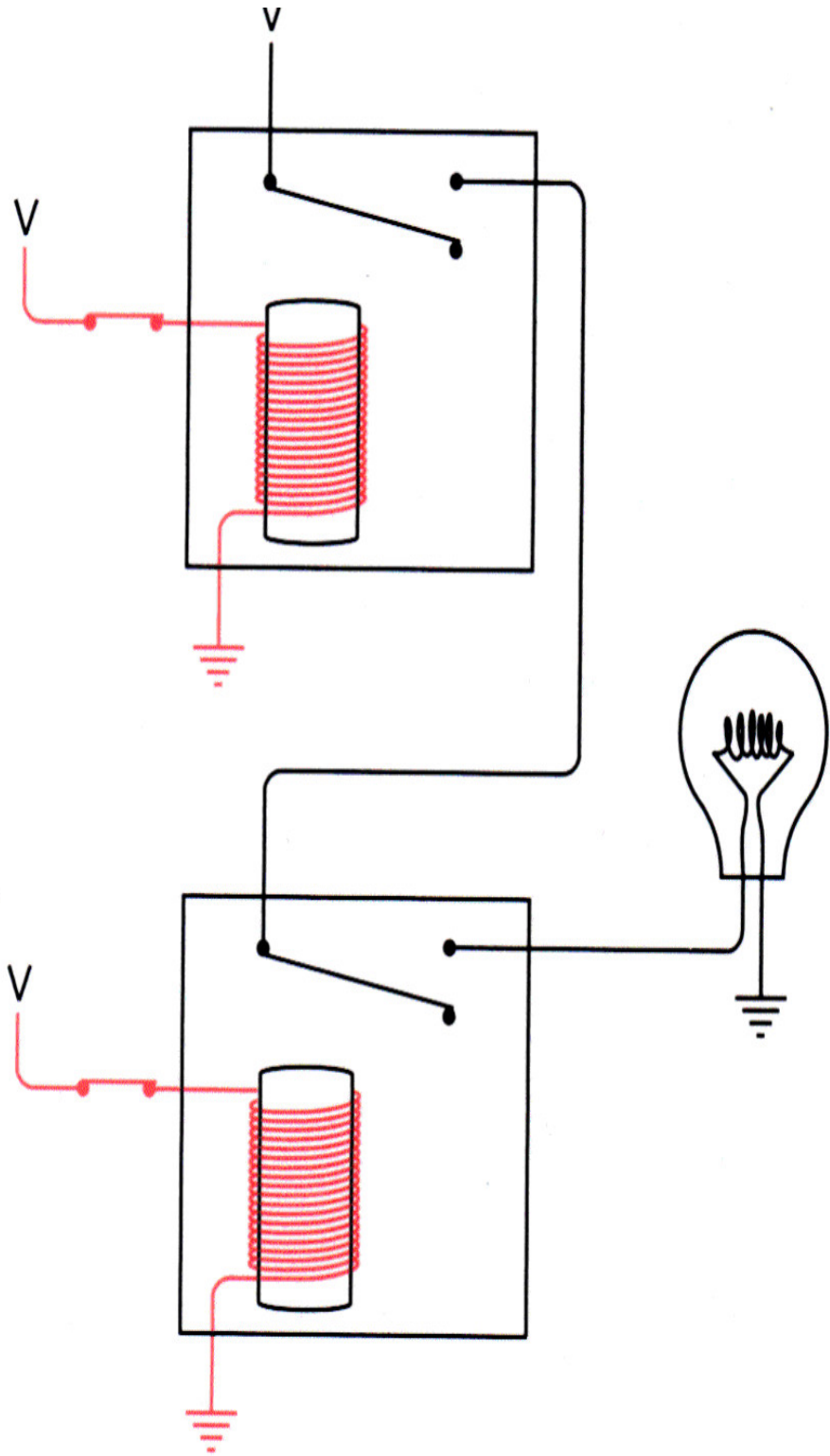
如果上面的开关闭合了，灯泡就会熄灭：



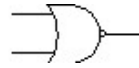
灯泡的熄灭是因为第二个继电器没有电源供应。同样，若下面的开关闭合灯泡也会熄灭：



若两个开关都闭合，灯泡还是不亮：



这种行为和或门的行为正好相反，被称为“NOR gate（或非门）”。下面是或非门的符号：



它和或门的符号很相像，只是在输出端有一个空心的小圆圈，这个小圆圈表示反向，故

而或非门也可用下面的表示：



或非门的输出如下表所示:

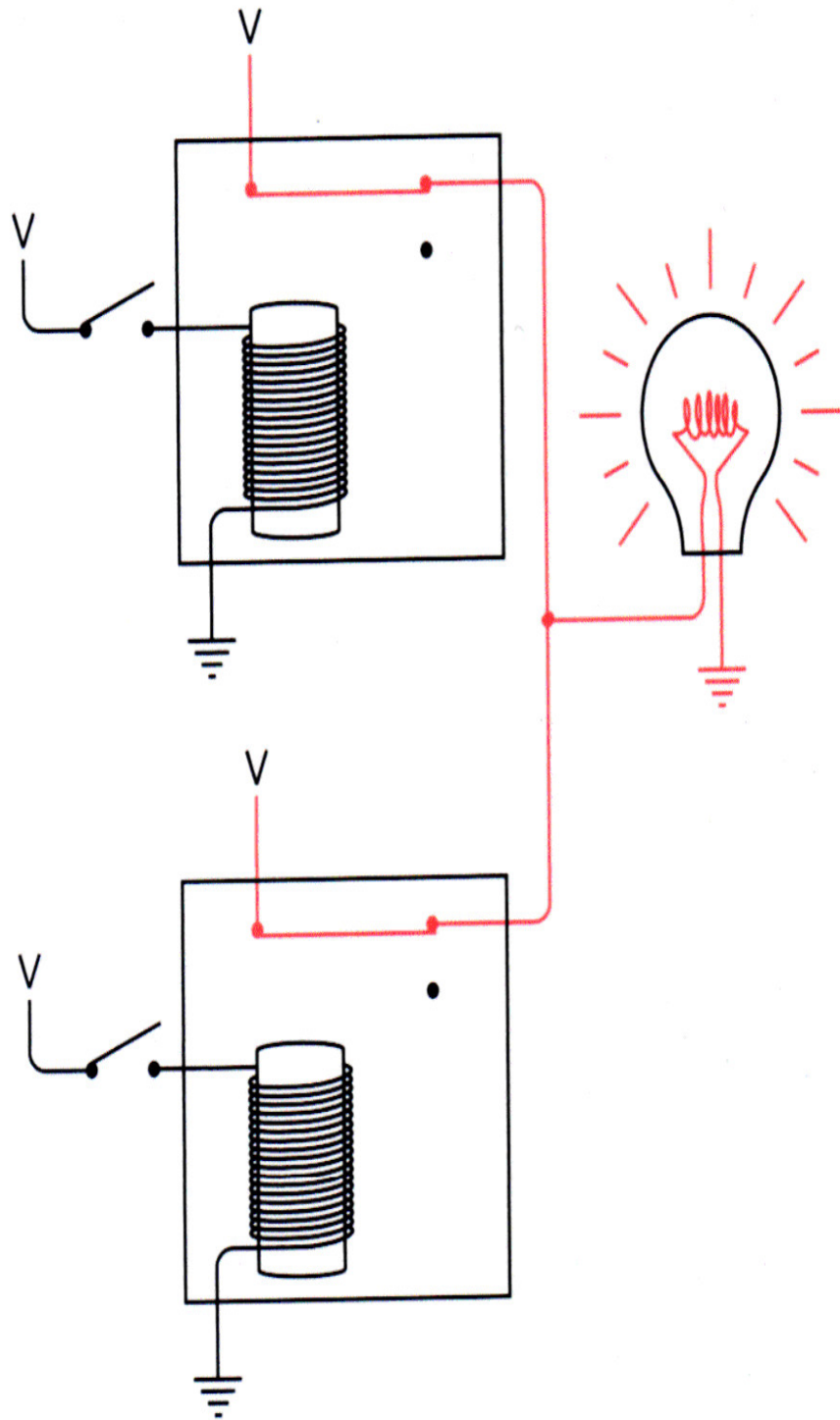
NOR	0	1
-----	---	---

0	1	0
---	---	---

1	0	0
---	---	---

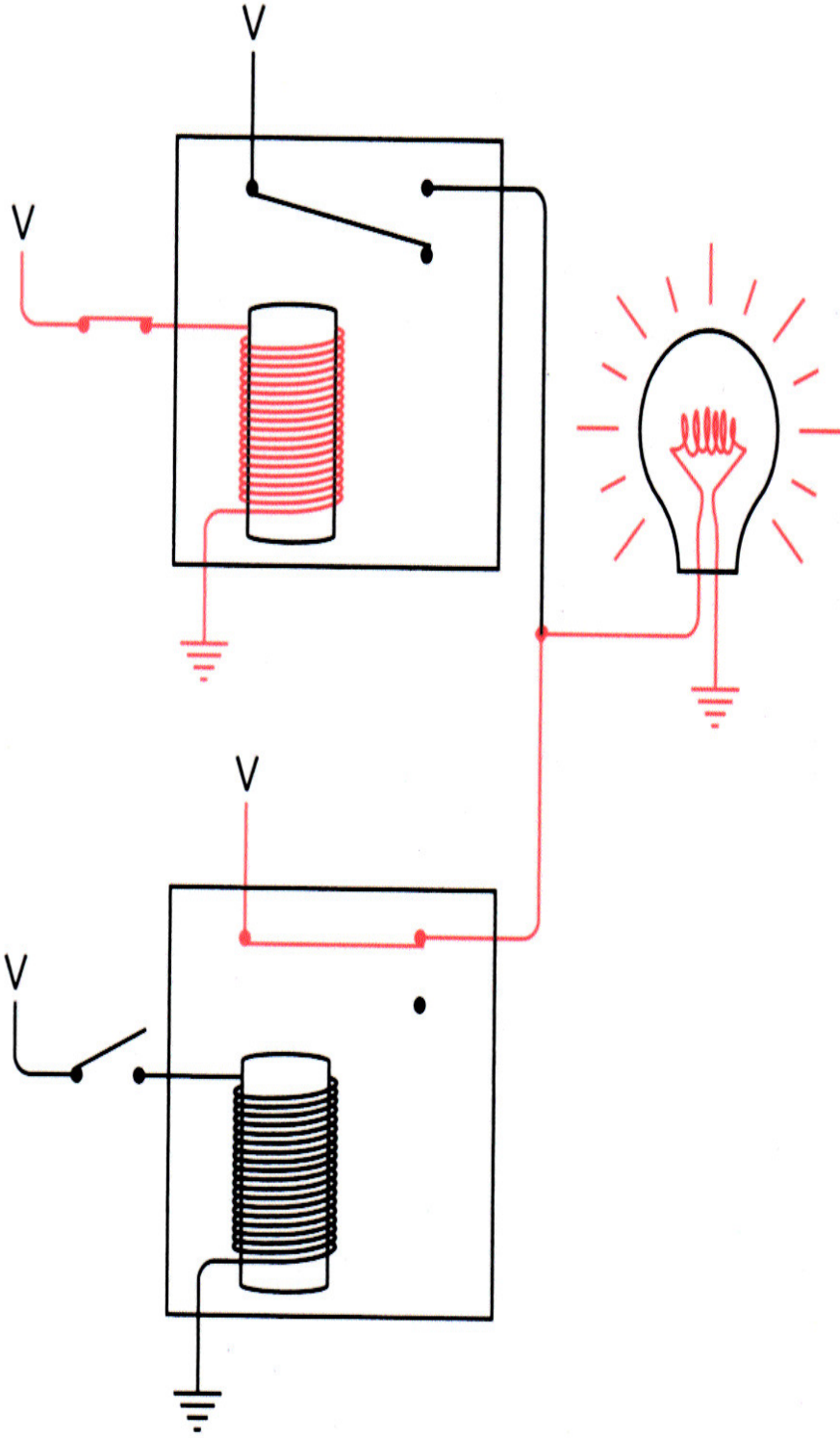
这张表显示的结果和或门相反。在或门中，输入端中只要有一个是 1，输出就是 1；只有输入端均为 0 时，输出才为 0。

连接两个继电器的另一种方式如下图所示：

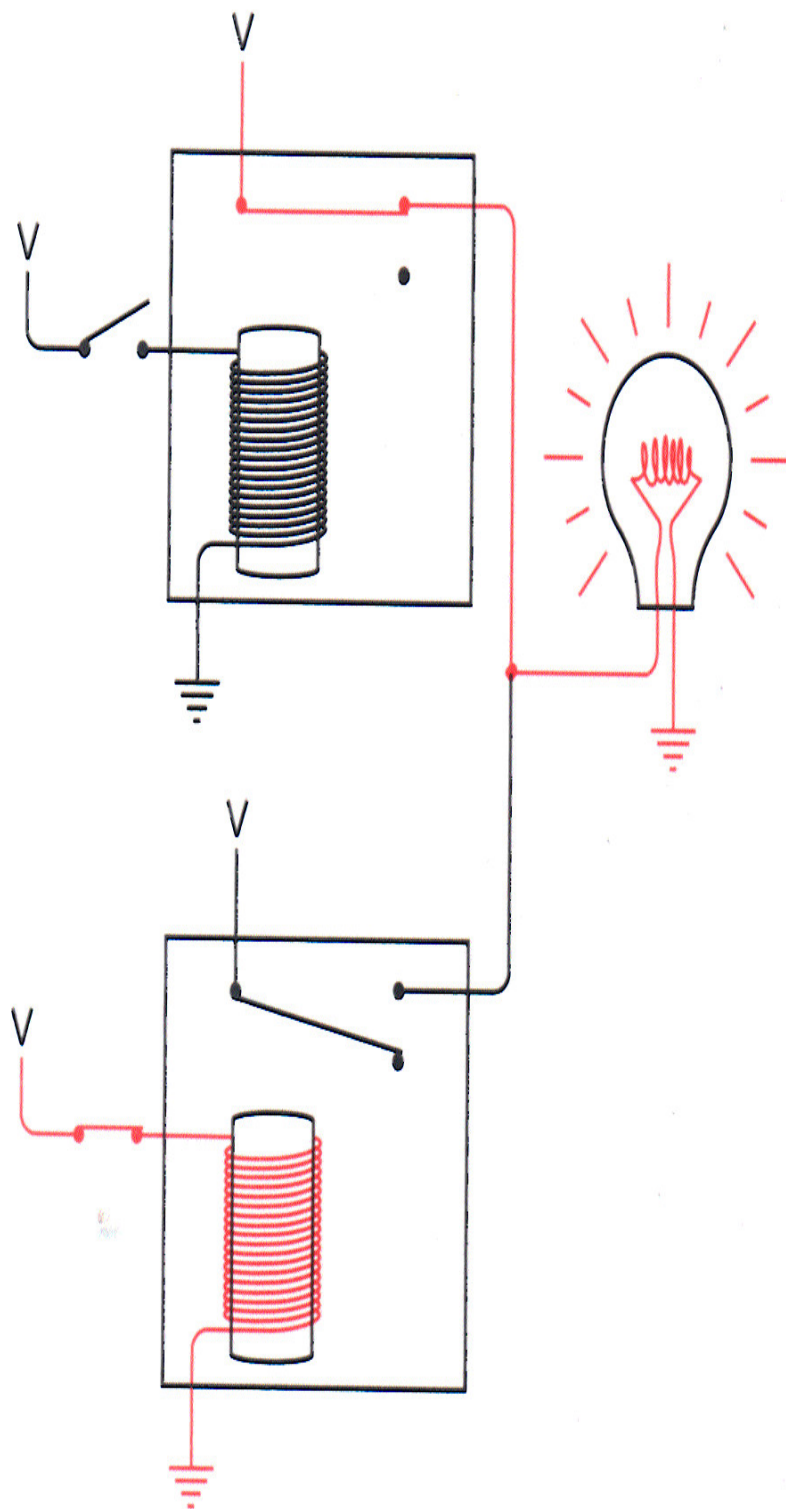


在这种情况下，两个输出连在一起。除了连在继电器的另一个触点上之外，这种连接形式与或门类似。当两个开关都断开时灯泡是亮的。

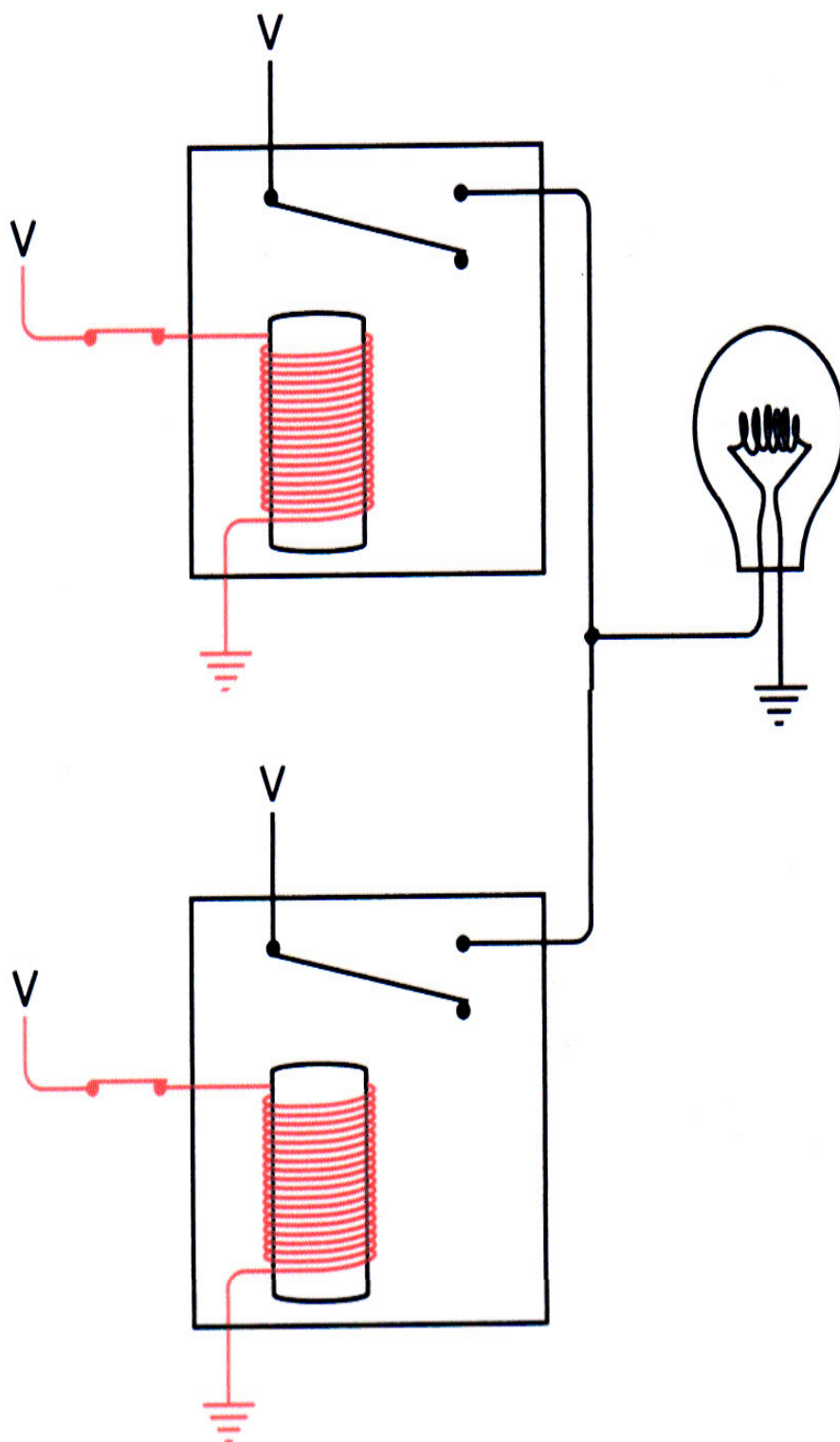
当只有上面的开关闭合时，灯泡也是亮的：



当只有下面的开关闭合时，灯泡也是亮的：



只有当两个开关都闭合时，灯泡才会熄灭：

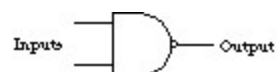


这种行为和与门的行为正好相反，被称为“NAND gate（与非门）”。与非门的画法和与门的画法很相像，只是在输出端加了一个小圆圈，表示其最后的输出和与门的输出是相反的：

输入	输出
----	----

与非门的输出如下表所示:

NAND	0	1
0	1	1
1	1	0

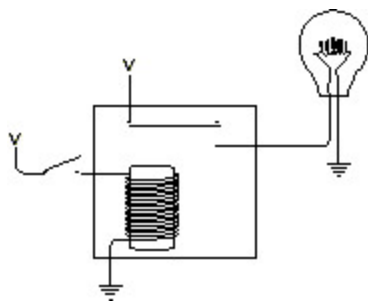


注意，与非门的输出与与门恰恰相反。对与门而言，当两个输入都为 1 时，输出才为 1； 否则输出就是 0。

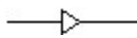
到此为止，我们已经看到可用四种不同的方式来连接有两个输入、一个输出的继电器， 每一种方式的行为功能都不一样。为避免画继电器，我们把这些连接称为逻辑门并使用电气 工程师们使用的符号来表示它们。特定的逻辑门的输出取决于其输入，总结如下：

AND	0	1	OR	0	1
0	0	0	0	0	1

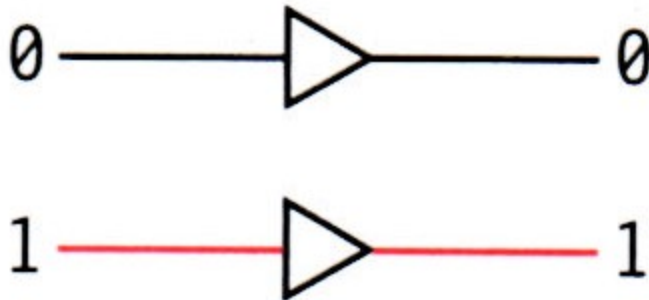
1	0	1	1	1	1
NAND	0	1	NOR	0	1
0	1	1	0	1	0
1	1	0	1	0	0



现在已有了四个逻辑门和一个反向器，完成这些工具的其实就是原始的继电器：



上图称为 缓冲器，用符号表示如下： 它和反向器的符号类似，只是没有小圆圈。缓冲器的特点是“什么都不做”，其输出和输入是相同的：



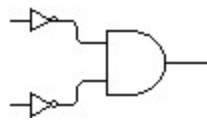
入是相同的：

当输入信号很弱时，可以使用缓冲器，这是因为这也正是多年前继电器被用于电报当中

的原因。此外，缓冲器也可用于延迟一个信号，这是因为继电器可能要求多一点儿动作时间，如1秒的几分之一才被触发。

本书从现在开始不再画继电器，取而代之的是电路将由缓冲器、反向器、4个基本逻辑门及更复杂的电路（如2-4译码器）组成。当然，所有这些部件也是由继电器构成的，但我们用不着看到它了。

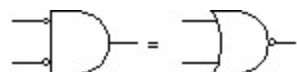
前面讲过，可用下面的小电路构造一个2-4译码器：



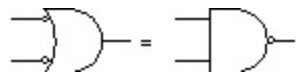
两个输入被反向后成为与门的输入。有时，像这样的配置可以去掉反向器而画成如下的样子：



注意与门输入端的小圆圈，这些小圆圈表示信号在这些点上被反向了，0会变成1，而1变为0。具有反向输入端的与门和或非门的行为是一样的：



只有两个输入端都为 0 时输出才为 1。同样，具有反向输入端的或门和与非门的行为是一样的：



只有输入端均为 1 时输出才为 0。这两对等同的电路实际上就是迪摩根定律的内容。迪摩根是维多利亚时代的另一位数学

家，他比布尔年长 9 岁。据说，他的书《Formal logic》发表于 1847 年，和布尔的《The Mathematical Analysis of logic》恰好是同一天。事实上，布尔正是由于受到发生在迪摩根和另一个英国数学家之间的剽窃事件的触动而研究逻辑的。（迪摩根最后证明是清白的。）很早以前，迪摩根就意识到了布尔思想的重要性。他无私地鼓励和帮助布尔进行研究，但最终除了他的这个著名的定律外，他几乎被人们遗忘了。

迪摩根定律可以简单地表示成：

$$\begin{aligned} \overline{A \times B} &= \overline{A} + \overline{B} \\ \overline{A + B} &= \overline{A} \times \overline{B} \end{aligned}$$

A 和 B 是两个布尔操作数。在第一个表达式中，它们被取反（即反向）后再相与。这和先 把它们相或后再取反（或非门的功能）的结果是一致的。第二个表达式中，两个操作数被取反后再相或，这和先 把它们相与后再取反（与非门的功能）的结果是一样的。

迪摩根定律对于简化布尔表达式，进而简化电路是一个很重要的工具。从历史上讲，这 正是香农的论文对电气工程师的真正含义。但是，专门简化电路并非本书的焦点，更重要的是让事物工作、起作用。下面我们要运行起来的就是一台简单的加法机。

第 12 章 二进制加法机

加法是最基本的算术运算。所以，如果想要建造一台计算机（这是本书隐含讨论的问题），必须首先知道如何构造一种机器，它可以把两个数加起来。当你解决了这个问题，你会发现加法正是计算机唯一所做的事情，因为通过使用用于加法的机器，我们还可以构造用加法来实现减法、乘法、除法以及计算房产抵押款、引导向火星发射卫星、下棋和电话计费等等功能的机器。

同现代的计算器和计算机比起来，本章构造的加法机庞大、笨重、速度慢且噪声大。但有意思的是构成它的部件完全是前几章学过的电子设备，如开关、灯泡、电线、电池以及可构成几种逻辑门的继电器。这个加法机包含的所有部件都于 120 年以前就已发明，而且，我们并不用真正地在屋子里建造它，只需在纸上和脑子里构造这台机器就行了。

这个加法机只能工作于二进制数，而且它缺少很多现代计算机（器）的辅助设备。它不能用键盘来敲入你想加的数，代之的你只能用一系列开关表示待加的数。它也不能用显示器显示结果，你所看到的只是一排灯泡。

但这台加法机确实实现了两数相加的功能，而且其工作方式和计算机做加法十分相似。二进制加法与十进制加法很像。当你相加十进制数如 245 和 673 时，你把问题分解成简单

的步骤，每一步只对一对十进制数字相加。本例中，第 1 步是把 5 和 3 加起来。生活中，你若能记住加法表，问题的解决就快多了。

十进制加法和二进制加法的一大区别是二进制数字的加法表要比十进制数字的加法表简单得多：

+	0	1
0	0	1
1	1	10

你可能在学校里记过上面这张表，并背诵过如下口诀：

0加0等于0，

0加1等于1，

1加0等于1，

1加1等于0，进1。

把相加结果的数前加上零，可以把加法表改写成如下形式：

+	0	1
0	00	01
1	01	10

这样一来，二进制数字相加的结果是两位数，分别称为“和”和“进位”（比如“1加1等于0，进位是1”）。现在，可以把这张二进制加法表分成两张表，第1张是表示“和”的表：

+和	0	1
----	---	---

0	0	1
---	---	---

1	1	0
---	---	---

第2张是表示“进位”的表:

+进位 0 1

0 0 0

1 0 1

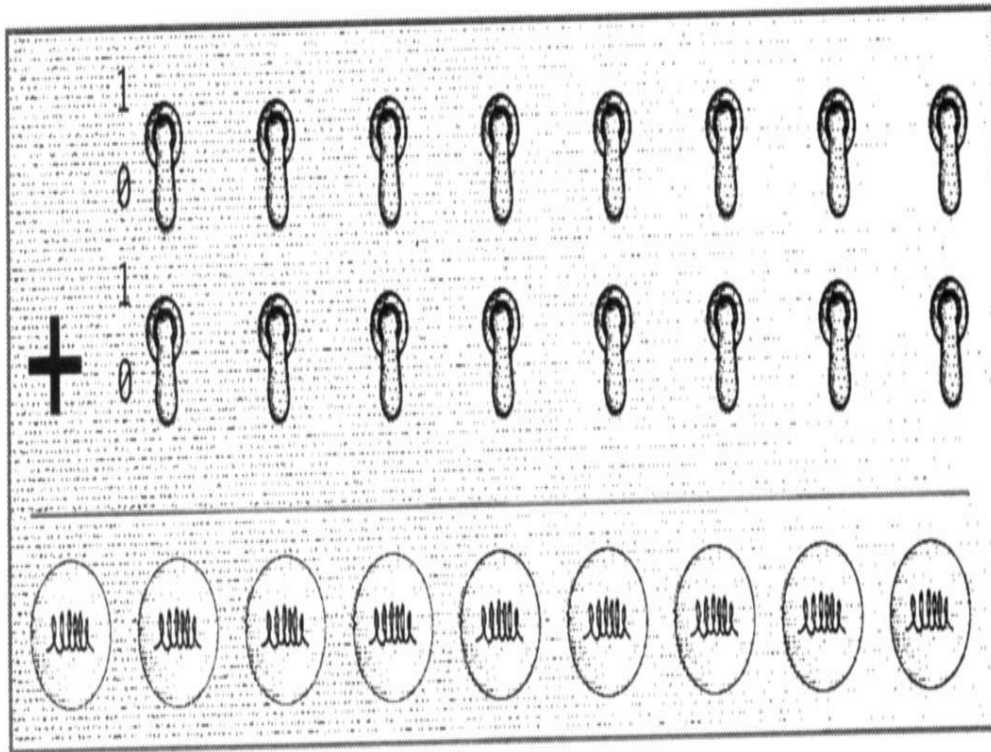
以这种方式来看待二进制加法就很方便了，因为加法机会分开求和与进位。构造二进制加法机需要设计一个能执行表中所描述操作的电路。因为电路的所有部件，如开关、灯泡、电线都是可以表示成二进制数的，因而该电路由于仅工作于二进制数从而大大降低了电路的复杂性。

与十进制加法一样，二进制加法也从最右边的一列开始，逐列相加两个数：

$$\begin{array}{r} 01100101 \\ + 10110110 \\ \hline 100011011 \end{array}$$

注意，当从右边加到第3列的时候，产生了一个进位。同样的情况也发生在第6、7、8列。我们要加多大的数呢？由于这个加法机只是在脑子里构造，因而可以加很长的数字。为更合理一些，选择不超过8位的二进制数。也就是说，操作数的范围是从0000-0000~1111-

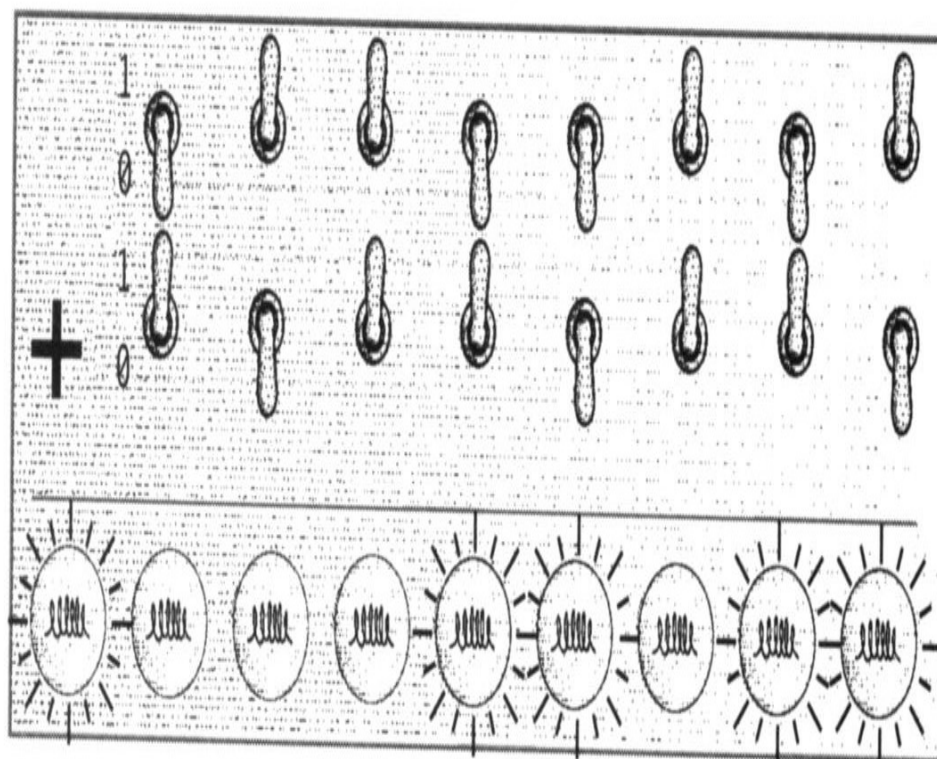
1111，即十进制的0~255。两个8位二进制数的和最大可以是1-1111-1110，即510。此二进制加法机的控制面板如下图所示：



板上有两行开关，每行 8 个。这些开关集是输入设备，我们将用它输入两个 8 位数。开关

往下表示 0，往上表示 1，正如家里墙上的开关。输出设备在板的底部，是一行灯泡，共 9 个。这些灯泡用来表示加法的结果，不亮的灯泡表示 0，亮的表示 1。我们用了 9 个灯泡是因为两个 8 位数相加的结果可能是 9 位数。

加法机的余下部分包含了以不同方式连接而成的逻辑门。开关触发逻辑门中的继电器，继电器接着点亮相应的灯泡。例如，如果我们想把 0110-0101 和 1011-0110 加起来（即前例中显示的两个数字），需把相应的开关设置成下面的样子：



灯泡的亮暗表明答案是 1-0001-1011。（当然，这只是希望的情况。毕竟，我们还没有把这个加法机构造出来！）

上一章提到过本书将会用到很多继电器，本章中的 8 位加法机就至少需要 144 个继电器，其中每一对数进行加法操作需要 18 个继电器（ $8 \times 18=144$ ）。如果画出完整的电路图，你一定会大惊失色，任何人都无法将连成一堆的 144 个继电器看得明明白白，所以我们将用逻辑门分步解决这个问题。

当你看到下面两个 1 位二进制数相加的进位表时，你可能立刻会想到逻辑门和二进制加法之间有某种联系：

+进位	0	1
0	0	0
1	0	1

你也许已意识到这和上章所述的与门的输出是一样的：

AND	0	1
0	0	0
1	0	1

所以，与门可以用来计算两个 1 位进制数位相加得到的进位。看来我们已取得一点儿进展了，下一步就要看看有没有继电器能完成下面的工作：

+和	0	1
0	0	1
1	1	0

这是二进制加法运算中的另一半问题，虽说表示和的这一位不如进位那么容易实现，但 我们会有办法。

首先应意识到或门的输出和我们所期望的很近似，只是右下角的结果不同：

OR	0	1
0	0	1
1	1	1

而对于与非门而言，除了左上角的输出不同以外，其他结果也与期望的一样：

NAND	0	1
0	1	1
1	1	0

所以，使用相同的输入，让我们把与非门和或门连接起来：



A 输入

B 输入

或门输出

与非门输出

下表总结了或门和与非门的输出，并将其结果和加法机所要求的结果进行比较：

A\输入	B输入	或门输出	与非门输出	需要的结果
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

注意，当或门和与非门的输出都为 1 时，就可以得到期望的结果 1，这暗示着把两个输出 作为与门的输入：



A 输入

B 输入

输出

好，这样就能满足要求了。整个电路仍然只有两个输入，一个输出。两个输入既连到了或门，也连到了与非门。或

门和与非门的输出作为与门的输入，从而得到预期的结果：

或门		与非门		与门
A输入	B输入	或门输出	与非门输出	与输出
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

这个电路有它自己的名字，称为“异或门（ Exclusive OR gate 或 XOR）”。异或门输出为 1 时，A输入为1或B输入为1，但不能同时为 1。不用再去画一个或门、一个与非门和一个与门， 可以用电气工程师规定的符号来表示它：



它看上去和或门很像，只是在输入端还有一条曲线。异或门的行为表示如下：

XOR		0	1
0	0	0	1
1	1	0	0

异或门是本书需要详细描述的最后 一个逻辑门（在电气工程中有时还会遇到第六个门，称为“同或门”，同或门只有两个输入相等时输出才为 1。同或门描述的输出情况正好和异或门相反，所以这个门的符号和异或门相同，同时在输出端有一个小圆圈）。

让我们来总结一下。两个二进制数相加产生两个表，一个是表示“和”的表，另一个是表示“进位”的表：

+和			+进位		
	0	1		0	1
0	0	1	0	0	0
1	1	0	1	0	1

用下面两个逻辑门可以得到同样的结果:

XOR	0	1	AND	0	1
0	0	1	0	0	0
1	1	0	1	0	1

二进制数的“和”可以由异或门得到，而“进位”可以由与门得到，所以可以把异或门 和与门结合起来来完成两个二进制数 **A**和**B**的加法：



A 输入

B 输入

和输出

进位输出

不用画与门和异或门，可以把上图简单地表示成如下的样子：



A 输入

半加器

和输出

B 输入 进位输出

其中的方块称为“半加器（Half Adder）”，它可以把两个二进制位 A 和 B 相加，从而得到一个和输出 (简称 S) 和一个进位输出 (简称 CO)。但大部分二进制数是多于 1 位的，半加器不能够把前一步的进位加到本次运算中。例如做如下加法：

$$\begin{array}{r} 1111 \\ + 1111 \\ \hline 11110 \end{array}$$

只能用半加器来计算最右边一列数：即 1 加 1 等于 0，进位为 1。对于右边第 2 列数，由于进位的存在，需要加 3 个数。接下来的几列都有这个问题，每一列二进制位的加法都包括了来自前一列的进位。

要把 3 个二进制数相加，需要按如下方式把两个半加器和一个或门连接起来：



进位输入 和输出

半加器

A 输入

B 输入

半加器

进位输出

要理解它的工作原理，先从最左边第一个半加器的 **A** 输入和 **B** 输入开始，其输出是一个 和 及相应的进位。这个和必须和前一列的进位输入 (简称**CI**) 加起来，然后把它们输入到第二个半加器。第二个半加器的和输出是最后的和。两个半加器的进位输出又输入到一个或门，或门产生了本次加法的进位输出。你可能会想这里还需要一个半加器，这当然是可行的。但

当你把所有的可能情况考虑完，你会发现两个进位不可能同时为 1。当两个输入不能同时为 1

时，或门已足够用于表示两个进位的加法，此时或门和异或门的功能是相同的。上图可简化表示为下面的方块图，称其为“全加器（Full Adder）”：



进位输入

A 输入 B 输入

全加器

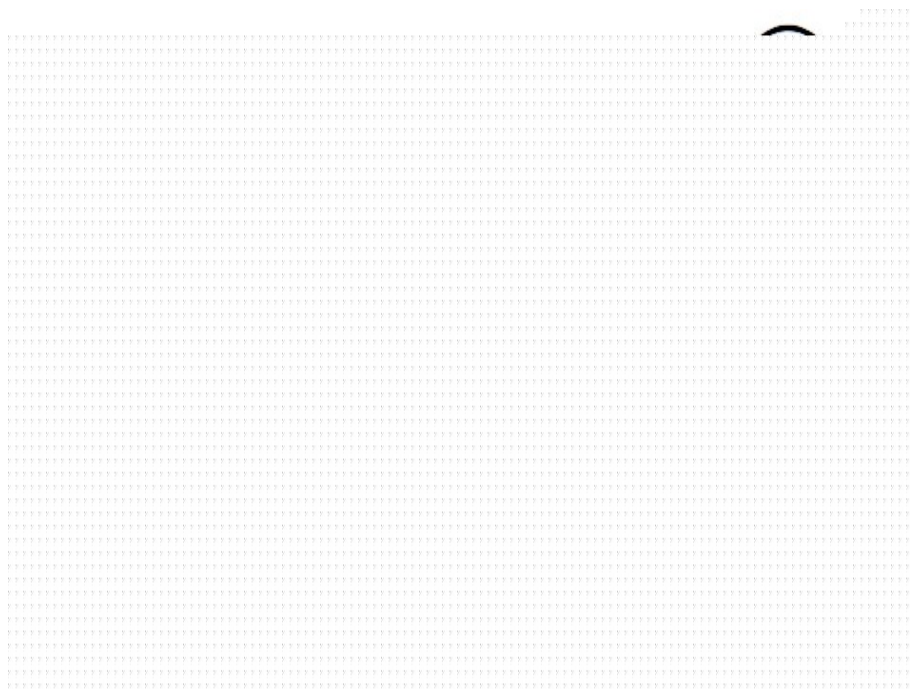
和输出 进位输出

下面的表是对全加器所有可能的输入及其相应输出的小结：

A输入	B输入	进位输入	和输出	进位输出
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

前面说过加法机需要 144个继电器，这个数目是如何得到的呢？每个与门、或门、与非门 都需要 2个继电器，所以，一个异或门需 6个继电器。一个半加器由一个异或门和一个与门构成，所以它要 8个继电器。1个全加器需要两个半加器和一个或门，所以它要 18个继电器。对于8位二进制加法机而言，共需 8个全加器，因而总共是 144个继电器。

回想一下本章最开始那个带开关和灯泡的控制面板：



现在可以把这些开关和灯泡连接成全加器了。首先把最右边的两个开关和一个灯泡连到一个全加器上，如下图所示：



全加器

进位输出

当把两个二进制数相加时，第 1 列的处理有所不同。因为接下去的几列可能包括来自前面加法的进位，而第 1 列不会有进位，所以全加器的进位输入端是接地的，这表示输入为“0”。第 1 列相加后很可能会产生一个进位输出，这个进位输出是下一列加法的输入。

对于接下去的两个二进制位和灯泡，可以按如下办法连接全加器：



进位输入

全加器

进位输出

第一个全加器的进位输出是第二个全加器的进位输入。接下去的每一列数都以这种方式 连接，每一列的进位输出都是下一列的进位输入。

第八个灯泡和最后一对开关连到最后一个全加器上，连接方式如下图所示：



进位输入

全加器

这里最后的进位输出连到第九个灯泡上。这样，8个全加器就构造成功了。

还可以用另一种方式来看 8个全加器的集成，每个全加器的进位输出都是下一个全加器的 进位输入：



进位输入

进位输出

下面是一个完整的屏蔽在一个盒子中的 8 位加法器。输入是 A 和 B
标识为从 A ~ A 及 B ~ B。

0 7 0 7

输出为和输出，标识为从 $S \sim S$:



A 输入

B 输入

进位输入

8 位加法器

进位输出

和输出

这是标识多位数字的常用方法。下标为 0 的位 A 、 B 和 S 表示最右边的、最不起眼的位。

0 0 0

而位 A 、 B 和 S 是最左边的、最引人注目的位。例如，下面展示的是这些字母是如何用来表示

7 7 7

二进制数 0110-1001的:

7 6 5 4

A

A

A

A

3 2 1 0

A

A

A

A

0 1 1 0 1 0 0 1

下标始于 0，且向高位递增的原因是它们和 2 的乘方数（幂）是对应的：

2_7 2_6 2_5 2_4 2_3 2_2 2_1 2_0

0 1 1 0 1 0 0 1

如果把每个二进制位和对应的 2 的幂次方相乘再依次相加，你就会得到 0110-1001 的十进制数表示，即 $64+32+8+1=105$ 。

8 位加法器的另一种画法是：



A 输入

B 输入

进位输入

8 位加法器

进位输出

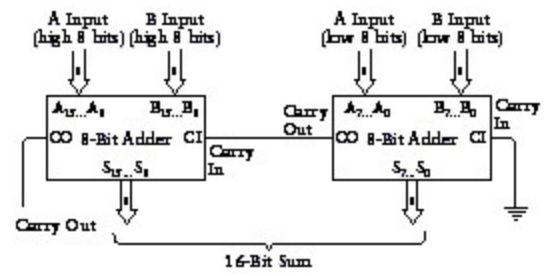
双线箭头包含了8个输入端，代表一组 8个分开的信号。它们标识为 $A \cdots A$ 、 $B \cdots B$ 、 $S \cdots S$

也用来表示一个 8 位二进制数。

7 0 7

0 7 0

一旦构造了一个 8 位加法器，就可以构造另一个加法器。把它们级联起来可以很容易地构成 16 位加法器：



A 输入

B 输入 A 输入 B 输入

(高 8 位)

(高 8 位) (低 8 位)

进位输出

8位加法器 8位加法器

进位输入

进位输入

进位输出

右边加法器的进位输出连到左边加法器的进位输入端。左边加法器的输入包含了两个加数的高8位，同时产生了结果的高8位。

现在，你可能会问：“计算机真的是以这种方式把数字加起来的吗？”基本上是这样的，但不完全是。首先，加法器应该做得更快。如果你明白这个电路是如何工作的，你会看到最低位相加

产生的进位作为下一列数相加的一个输入，而第3列的加法又等着第2列加法的进位，依此类推。加法器总体的速度等于加数的位数乘以单个全加器的速度。这种进位方式称为行波进位。更快的加法器使用称为先行进位的加法电路，从而加快了加法进程。

第二（但是十分重要），计算机再也不用继电器了！尽管它们曾经用过。建于20世纪30年代初的第一批数字计算机使用继电器，后来又用了真空管。现代计算机用晶体管。当用在计算机中时，晶体管和继电器的功能差不多，但是晶体管速度更快，体积更小，更安静，更省电，而且还便宜不少。构造一个8位加法器仍然需要144个晶体管（如果采用先行进位，则需要更多），但整体电路的体积却小多了。

第 13 章 如何实现减法

在你确信继电器可以连接起来以构成二进制加法器后，你可能会问：“减法器如何实现呢？”本章将会为你解答这个问题，且提出这个问题也表明你有了一定的理解力。减法和加法在某些方面是互为补充的，但两种计算的机制不同。加法从最右边一列向最左边一列计算，每一列的进位都加到下一列中去。减法不用进位，相反，要用到借位——一种本质上与加法不同的机制。

例如，让我们看一道典型的不断借位的减法题目：

$$\begin{array}{r} 253 \\ - 176 \\ \hline ??? \end{array}$$

要做这道题，从最右边一列开始。首先，6比3大，所以需要从5借1，这样就变成了13减6，结果是7。由于从5借了1，5就变成了4，4比7小，所以继续从2借1，14减7等于7。2被借1后成为1，1减1为0，所以最后结果是77：

$$\begin{array}{r} 253 \\ - 176 \\ \hline 77 \end{array}$$

如何用逻辑门来实现这看似不合常理的逻辑呢？我们不会直接用这种方法，代替的是用一个小技巧，使不通过借位来实现减法。这会是

一个使大家都满意的好办法。详细地了解减法的完成是很有用的，因为它和用二进制编码在

计算机中存储负数的机制有很大联系。为解释这样的工作，需要清楚地指明两个操作数，即减数和被减数。减数从被减数中去

掉后，结果是二者之差：

被减数

— 减数 差

要想不借位，首先将减数从 999 中减去：

999

−176

823

这里用 999 是因为操作数是 3 位，如果是 4 位数，就用 9999。把一个数从一串 9 中减去得到的结果称为 9 的补数或补码。176 的 9 的补数是 823，反之，823 的 9 的补数是 176。这样做的好处在于，无论减数是什么，计算 9 的补数永远不需要借位。

在计算出减数的 9 的补数之后，把它加到原来的被减数上：

最后，你再加 1 并且减去 1000:

2 5 3

+ 8 2 3

1 0 7 6

1 0 7 6

+ 1

— 1 0 0 0

7 7

这样就得到结果了。答案和以前一样，且你根本不用借位。这是什么原理呢？原来的减法题目是：

$$253 - 176$$

表达式加一个数再减同一个数得到的结果是一样的。所以先加上 1000，再减去 1000：

$$253 - 176 + 1000 - 1000$$

这个式子等同于下面的式子： 再按如下方式重新组合：

$$253 - 176 + 999 + 1 - 1000$$

$$253 + (999 - 176) + 1 - 1000$$

这与前面描述过的用 9 的补数进行的计算是一致的。虽然用了两个减法和两个加法来代替一个减法，但是也因此省去了讨厌的借位。

但是，如果减数比被减数大怎么办呢？例如如下计算：

$$\begin{array}{r} 176 \\ -253 \\ \hline ??? \end{array}$$

通常情况下，你看到这个式子后可能会说：“减数比被减数大只需交换两数位置，再做减法，然后给结果取个相反数。”于是你在脑子里交换了它们的位置，并求出了答案：

$$\begin{array}{r} 176 \\ -253 \\ \hline -77 \end{array}$$

要省去借位来做这道题和前面的例子有所不同。首先你要求出 253 的 9 的补数，即

$$\begin{array}{r} 999 \\ -253 \\ \hline 746 \end{array}$$

再把该补数和原来的被减数相加：

1 7 6

+ 7 4 6

9 2 2

这时候，按照上一道题的步骤，你应该对其加 1 再减去 1000，但在本题中，这种方法不会生效。如果你还按这种步骤做，就需要从 923 中减去 1000，这又导致了借位。

既然实际上前面已经加了 999，这里再减去 999：

$$\begin{array}{r} 922 \\ -999 \\ \hline ??? \end{array}$$

当做到这一步时，可看出结果是个负数，故需要交换两数位置，不过这样再做减法时已不需要借位，答案如预期所料：

$$\begin{array}{r} 922 \\ -999 \\ \hline -77 \end{array}$$

同样的方法可用于二进制数减法，而且会比十进制数减法来得简单。让我们看看该如何做。原来的减法题目是：

$$\begin{array}{r} 253 \\ -176 \\ \hline ??? \end{array}$$

当把这些数转化为二进制数时，问题变成：

$$11111101$$

— 1 0 1 1 0 0 0 0

? ? ? ? ? ? ? ?

步骤1 用11111111减去减数:

1 1 1 1 1 1 0 1

— 1 0 1 1 0 0 0 0

0 1 0 0 1 1 1 1

当计算十进制数减法时，减数是从一串 9 中减去，得到称为 9 的补数的结果。对于二进制数减法，减数从一串 1 中减去，差称为 1 的补数。但请注意，求 1 的补数实际上并不需要做减法，因为 1 的补数中，原来的 0 变成 1，原来的 1 变成 0，所以，1 的补数有时也称为相反数或反码。

（你是否还记得第 11 章中反向器的作用是把 0 变成 1，把 1 变成 0。）

步骤 2 把步骤 1 中求得的补数和被减数相加：

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1 \\ +\ 0\ 1\ 0\ 0\ 1\ 1\ 1\ 1 \\ \hline 1\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 0 \end{array}$$

步骤3 对结果加 1:

步骤4 减去100000000 (256) :

1 0 1 0 0 1 1 0 0

+ 1

1 0 1 0 0 1 1 0 1

该结果就是十进制数 77。

101001101

— 100000000

1001101

现在把两数颠倒位置后再做一遍。在十进制中，减法题目对应于：

$$\begin{array}{r} 176 \\ -253 \\ \hline ??? \end{array}$$

而在二进制中，即是：

1 0 1 1 0 0 0 0

− 1 1 1 1 1 1 0 1

? ? ? ? ? ? ? ?

步骤1 从11111111中减去减数。得到补数:

$$\begin{array}{r} 11111111 \\ - 11111101 \\ \hline 00000010 \end{array}$$

步骤2 把步骤1中的补数和被减数相加:

$$\begin{array}{r} 10110000 \\ + 00000010 \\ \hline 10110010 \end{array}$$

现在, 11111111 必须再从结果中减掉。当减数比被减数小时, 可以通过先加 1再减去

100000000来达到此目的。但现在这样做却会用到借位。所以, 我们先用 11111111减去步骤 2

中的结果:

$$\begin{array}{r} 11111111 \\ - 10110010 \\ \hline 01001101 \end{array}$$

这实际上是对步骤 2中得到的结果取反。最后的结果是 77, 而真正的答案应该是一 77。现在, 已经可以改进加法机使它既能执行加法操作亦能执行减法操作。为使简便起见,

这个加/减法机只执行被减数大于减数的减法操作, 即差为正数的操作。该加法机的核心部件是由逻辑门集成的 8位全加器:



A 输入 B 输入

进位输入

8 位加法器

进位输出

和输出

前面讲过输入 $A \sim A$ 及 $B \sim B$ 连接到开关上，用于表示 8 位操作数。进位输入端接地。

0 7 0 7

$S \sim S$ 连接 8 个灯泡，用于表示加法的和。由于和可能会是 9 位数，进位输出端也连了一个

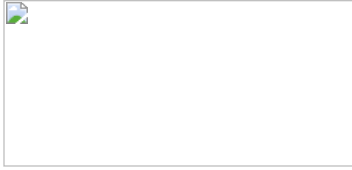
0 7

灯泡。

控制面板如下图所示：



上图中，开关被设为 183（或 10110111）和 22（或 00010110），产生的结果是 205 或 11001101。用于加/减法的新的控制面板有一点儿修改，它包含了一个用于选择做加法还是做减法的额外开关。



减法

加法

上溢 下溢

如图所示，当这个开关向下时表示选择加法运算，反之是选择减法运算。此外，只有最右边的 8 个灯泡用于表示结果，第九个灯泡用来标识上溢 / 下溢，它指明了一个不能用 8 个灯泡表示的数。当加法操作得到的和大于 255（称为上溢）或减法计算中出现一个负数（下溢）时，这个灯泡就会亮。减数比被减数大时，结果就是一个负数。

这个加法机主要增加了一个求 8 位二进制数的补数的电路。由于一个数的补数就是取其每一位的相反数，所以这个电路看起来很简单，就是 8 个反向器而已。



输入

输出

该电路存在一个问题，就是它不分情况地对输入求反。我们需要一台既能做加法又能做减法的机器，而此电路只有做减法时才取反。对它进行一下改进，如下图所示：



输入

取反

输出

图中标识为“取反”的信号输入到每一个异或门中。回忆一下异或门的功能：

XOR	0	1
0	0	1
1	1	0

如果“取反”信号为 0，则异或门的 8 个输出和 8 个输入是相同的。
例如，如果输入是

0 11 0 0 0 0 1，则输出也是 0 11 0 0 0 0 1；若“取反”信号为 1，则输出取反。例如，当输入是

01100001时，输出为 10011110。让我们把 8个异或门集成到一个盒子里，称为求补器：

7

6



输入 输入

取反

输入 输入

输入 输入

输入 输入

求补器

1
0
5
4
3
2
1
0
5
4
3
2

输出 输出

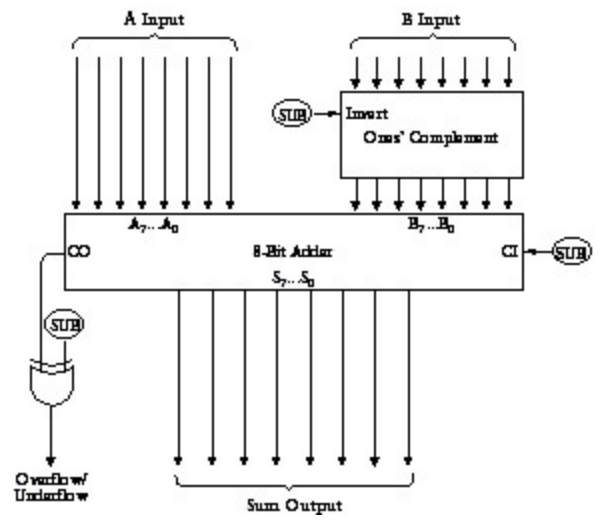
7

6

输出 输出

输出 输出 输出 输出

求补器、8位加法器及一个异或门可按下图连接：



A 输入

B 输入

取反

求补器

8位加法器

上溢

下溢 和输出

注意上图中有 3 个信号都标识为“SUB”，这是加 / 减法转换开关。当该信号为 0 时做加法，为 1 时做减法。做减法时，B 输入在送入加法器之前先求补。此外，做减法时，通过设置加法器的进位输入端 (CI) 为 1，使由加法器得到的结果加 1。对加法而言，求补电路没有起作用，CI 输入也就是 0。

“SUB”信号及加法器的 CO 输出作为异或门的输入来控制表示上溢 / 下溢的小灯泡。如果 “SUB”信号为 0（表示做加法），则当 CO 输出为 1 时灯泡点亮，这表示加法的和大于 255。

当做减法时，如果被减数大于减数，则加法器的 CO 端正常输出 1，这表示在减法的最后一步中要减去 100000000。所以，只有当加法器的 CO 输出为 0 时，上溢 / 下溢灯泡才被点亮。这时减数大于被减数，差是个负数。上面这个加 / 减法器现在还不能表示负数。

你一定兴致勃勃地想知道该如何实现减法了。本章一直在谈论负数，但没有指出二进制负数的表示方法。你可能会认为它的表示和十

进制负数一样，只需在数的前面加个负号。例如，-77 在二进制中写成 -1001101。你当然可以这么表示，但别忘了用二进制数的目的在于只用 0 和 1 表示所有的东西，当然也包括一个小小的负号了。

你可以用某一位代替负号，当该位为 1 时就表示负数，为 0 时表示正数，这似乎也是可行的。但还有一种方法，它不仅能表示负数，而且还很适于把正数和负数相加到一起。这种方法的不足之处是你必须提前决定数字需要多少位。

通常用来表示正、负数的方法的好处是这种方法能表示所有的正数、负数。我们把 0 想象成向一个方向延伸的无穷的正数流和向另一个方向延伸的无穷的负数流的中点：

$\dots -1\ 000\ 000 \text{ } -999\ 999\dots$

$-3 \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad 3\dots$

$999\ 999 \quad 1\ 000\ 000\dots$

但是,如果并不需要无限大或无限小的数,而是完全可以确定计算中所遇到的数的范围,情况便有所不同了。

下面来看看帐户的例子，人们有时可以在帐户上看到负数。假设帐户上从来没有超过

\$500的存款，而银行给我们的预支额是\$500，这就意味着帐户上的数字在\$499~-\$500之间。假设我们不会一次取出\$500，也不会写一张超过\$500的支票，同时我们只处理美元，而不考虑到更小的货币单位—美分。

这些假设表明帐户能处理的数字范围是从 $-500 \sim 499$ ，总共 1000 个数。这个限制暗示我们只能用 3 位十进制数，且可不用负号来表示这 1000 个数。其中的关键在于我们不需要 $500 \sim 999$ 之间的正数，所以它们就可以用来表示负数。下面是其工作原理：

用 500 表示 $\overline{\quad}$ 500 用 501 表示 $\overline{\quad}$ 499 用 502 表示 $\overline{\quad}$ 498

• • •

用 998 表示 $\overline{\quad}$ 2 用 999 表示 $\overline{\quad}$ 1 用 000 表示 0 用 001 表示 1 用 002 表示 2

•
•
•

用 497 表示 497

用 498 表示 498 用 499 表示 499

换句话说，以 5、6、7、8、9 开头的 3 位数实际上都表示负数。不用如下的表示法：

— 500 — 499 — 498 … — 4 — 3 — 2 — 1 0 1 2 3 4 … 497 498 499

而用这样的表示法：

500 501 502 … 996 997 998 999 000 001 002 003 004 … 497
498 499

注意这样形成了一个环形排序，最小的负数（500）看上去是最大的正数（499）的延续。数字 999 是比零小的第一个负数。如果给 999 加上 1，通常得到 1000。但由于只处理 3 位数，所以实际上是 000。

这种处理称为 10 的补数。要把 3 位负数转换成 10 的补数，需从 999 中减去它再加 1。换句话说，10 的补数是 9 的补数再加 1。例如，要把 — 255 写成 10 的补数，应先从 999 中减去 255 得到 744，再加上 1 后得到 745。

你可能听说过“减法不过是负数的加法”，你也可能回答过“其实还是不得不做减法”。

然而，通过使用 10 的补数，就不用去做减法了，全部都可以用加法来计算。

假设你有余额为 \$143 的帐户，并写了一张 \$78 的支票，这表明你要把 — 78 加到 143 上。— 78 的补数是 $999 - 78 + 1$ ，即 922。所以新的余额是 $143 + 922$ （忽略上溢），即 65。若我们再写一张 \$150

的支票，则必须减去 150，用补数表示就是850。先前的余额065加上850等于915，所以，新的余额实际上是一 \$85。

二进制中对应的系统称为 2 的补数。假设我们用 8 位二进制数工作，范围从 00000000～

11111111，对应于十进制的 0～255。这时如果你想要表达负数，则以 1 开头的每个 8 位数都表示一个负数，如下所示：

二进制数	十进制数
10000000	— 128
10000001	— 127
10000010	— 126
10000011	— 125
⋮	
11111101	— 3
11111110	— 2
11111111	— 1
00000000	0
00000001	1
10000010	2
⋮	
01111100	124
01111101	125

01111110	126
----------	-----

01111111	127
----------	-----

你可以表示的数的范围从 $-128 \sim 127$ 。最左边的一位称为符号位，1表示负数，0表示正数。

要计算2的补数得先求出1的补数再加上1，这等同于先求反再加1。例如，十进制数125是

01111101，要用2的补数来表示 -125 ，可先取反得10000010，再加1就得到10000011。可用上表来验证这个结果。要回到原来的数只需同样的操作：取反后加1。

这个系统使不用负号就能表示正、负数，它也使我们只用加法规则就可以随意进行正、负数运算。例如，计算 $-127+124$ ，利用上表即得

$$\begin{array}{r} 10000001 \\ + 01111100 \\ \hline 11111101 \end{array}$$

和是十进制的 -3 。

这里要注意上溢或下溢，即结果大于127或小于 -128 的情况。例如，125加125：

$$\begin{array}{r} 01111101 \\ + 01111101 \\ \hline 11111010 \end{array}$$

因为最高位是1，结果代表一个负数： -6 。再看 -125 加上它自己：

$$\begin{array}{r}
 10000011 \\
 + 10000011 \\
 \hline
 100000110
 \end{array}$$

由于限制了只取 8 位数，所以最左边的 1 被扔掉，剩下的 8 位表示 6。一般而言，若两个操作数的符号相同，而结果的符号与操作数的符号不相同，这样的

加法是无效的（即加法运算产生了溢出！）。现在，二进制数可以有两种不同的使用方法。二进制数可以是无符号的或有符号的，无

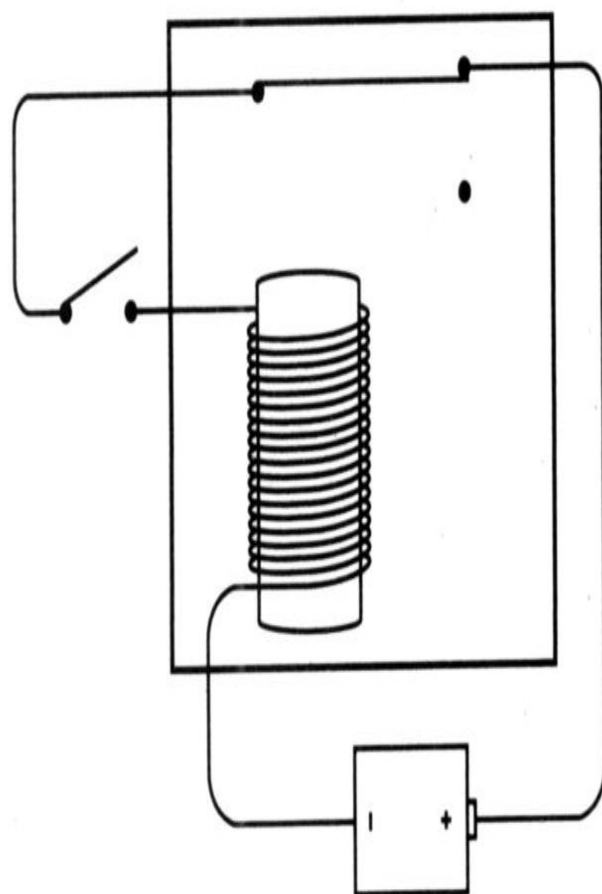
符号的二进制 8 位数的表示范围从 0~255，有符号的二进制 8 位数的表示范围从 -128~127。这些数本身不会告诉你它们是否带有符号。例如，假设有人问：“10110110 对应于十进制数的几？”这时，你必须先问清楚它是无符号数还是有符号数？它可能是 182 或 -74。

这就是二进制数的麻烦：它们仅仅是一些 0 和 1 而没有告诉它们的任何含义。

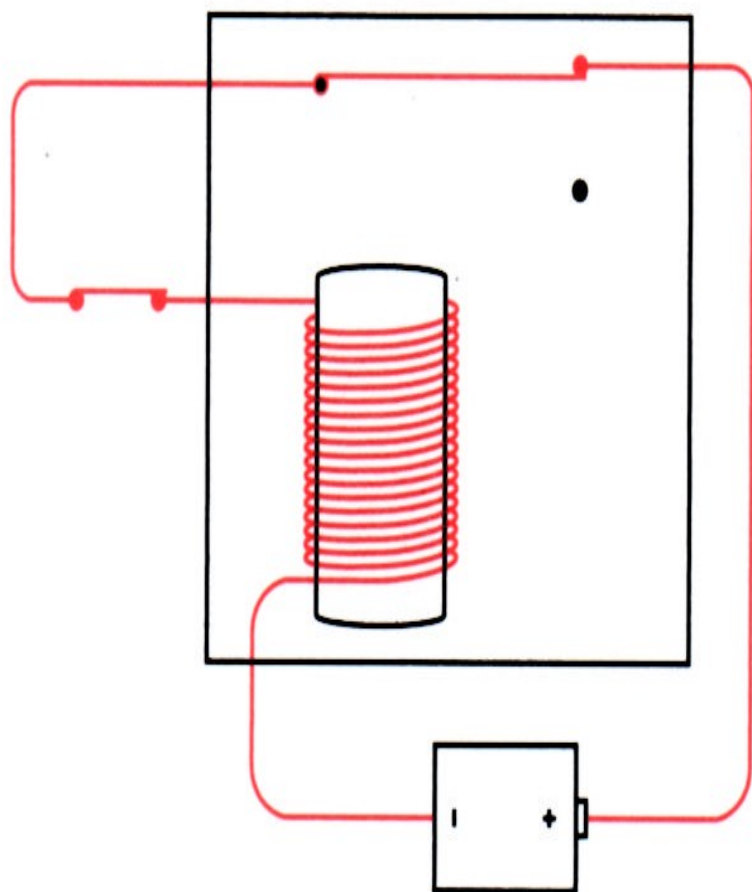
第 14 章 反馈与触发器

人人都知道电可以使物体运动。随便看一眼就会发现，很多家用电器中都装了电动机，如钟、风扇，食品加工机、CD机等等。电也能使扬声器中的磁芯振动，从而使音响设备、电视机产生了声音、话音和音乐。不过，电使物体运动的一个最简单、最神奇的例子可能是电子蜂鸣器和电铃。

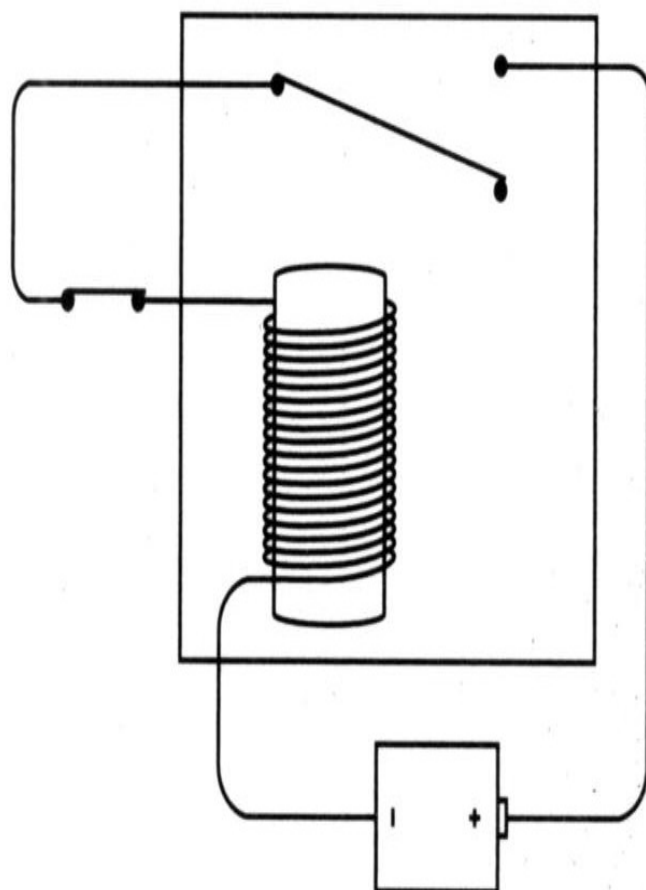
将继电器、电池、开关按如下形式连接：



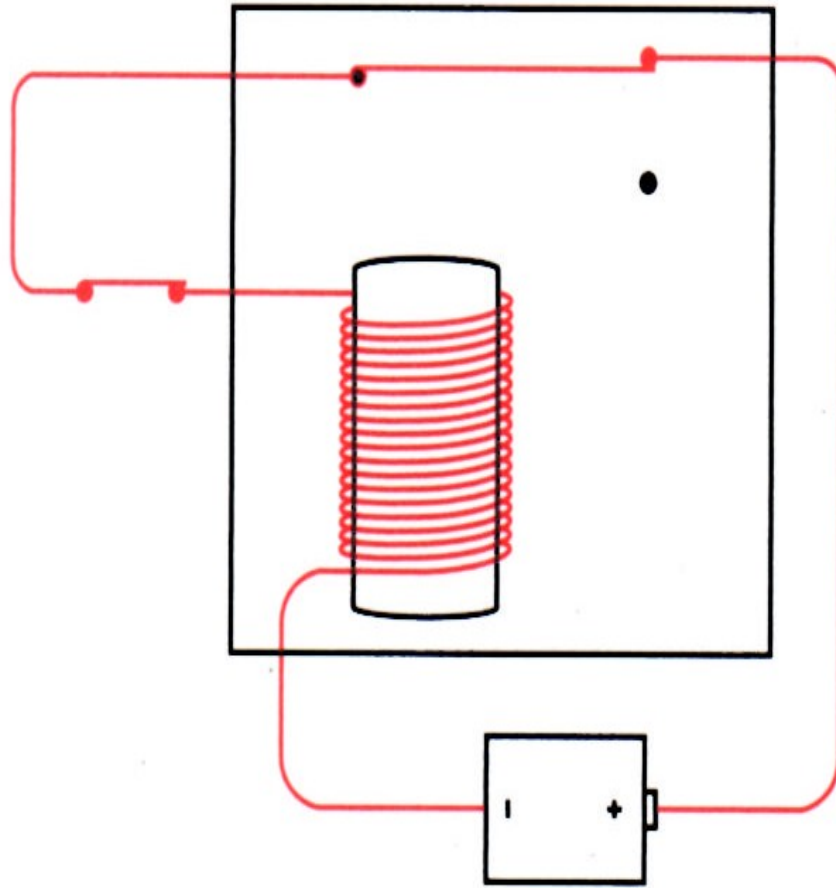
如果你觉得它看起来很奇怪，则你还没有发挥出你的想像力。我们还从未见过如此连接的继电器。原来的继电器中，输入和输出通常是分开的，这里却构成一个闭环。当闭合开关时，电路连通了：



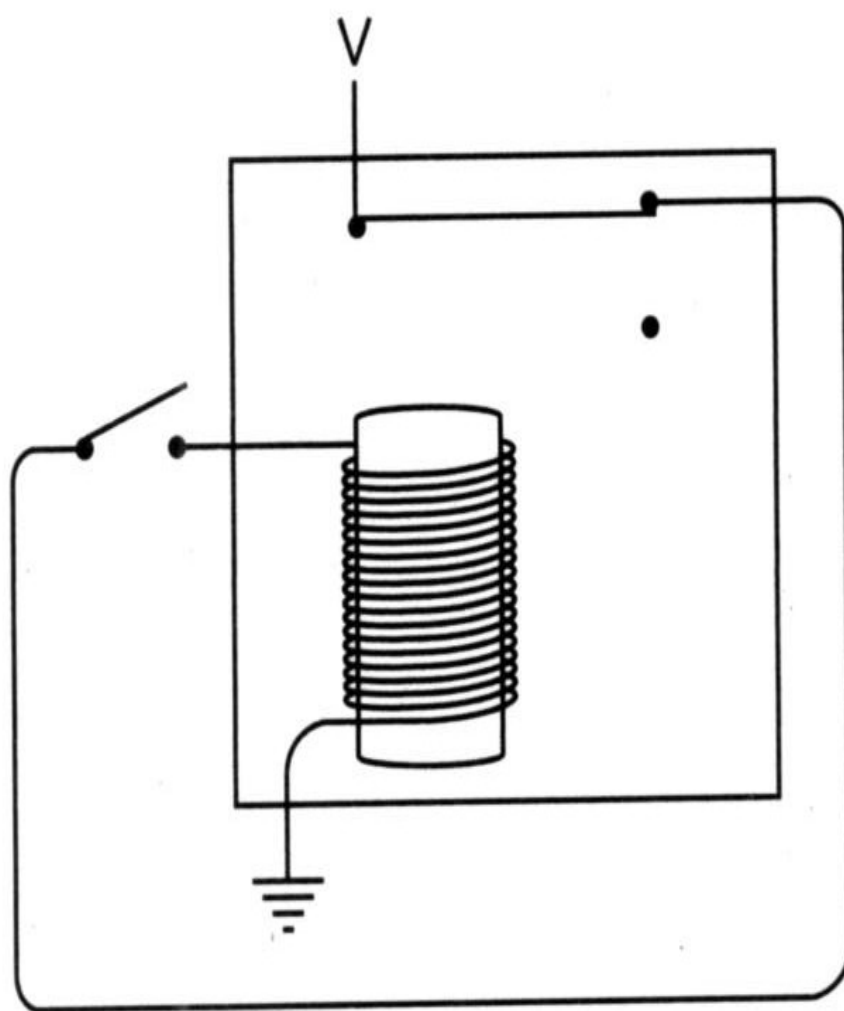
接通的电路使电磁铁把金属簧片拉下来（电流的作用）：



当金属簧片改变位置后，电路不再完整，电磁铁失去了磁性，金属簧片又弹回原来的位置：

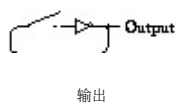


这样，电路便又一次接通了。可见，只要开关是闭合的，金属簧片就会上下跳动——使电路闭合或断开——并制造一种声音。如果金属簧片制造了一种刺耳的声音，它就构成了一个蜂鸣器。如果金属簧片附上一把小锤子，再加一个金属锣，它就构成了一个电铃。

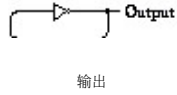


有两种方法可用来连接继电器以构造一个蜂鸣器，下面是另一种方法的描述：

你可能从上述图中认出了这是第 11 章介绍过的反向器，所以电路可以简化为：



对于反向器而言，当输入为 0 时，输出为 1；输入为 1 时，输出为 0。在该电路中闭合开关 会使反向器中的继电器间断地闭合和断开。如果去掉开关，可以使反向器连续地工作，如下 图示：



这幅图似乎在演示一种逻辑矛盾，反向器的输出是和其输入相反的，但是在这里，其输出同时又是其输入。需要特别指出的是，反向器实际上是一个继电器，而继电器从一个状态转换到另一个状态是需要时间的。所以，即使输入和输出是相等的，输出也会很快地改变，成为输入的倒置（当然，随即输出也就改变了输入，如此反复）。

电路的输出是什么呢？其实就是提供电压和不提供电压之间的变换。或者说输出要么是0，要么是1。

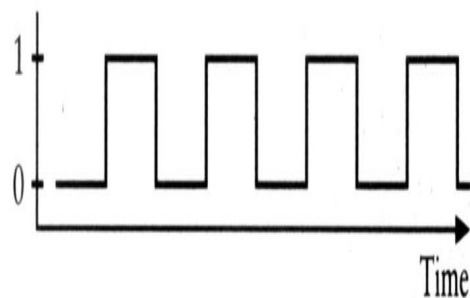
这个电路称为 振荡器，它和我们以前见到的每样东西都有本质上的区别。以前，所有的电路都靠手动地断开或闭合开关来改变状态，而振荡器却不需要人的干涉，它可以自主地工作。

当然，单独的一个振荡器不会有什么用，但在本章的后面及接下去的几章里，你会看到这个电路和其他电路连接后构成了自动控制中一个十分关键的部分。所有计算机都靠某种振荡器来使其他部件同步工作。



振荡器的输出是 0 和 1 的交替序列，可以用下图形象地来表示它：

图中，水平轴表示时间，垂直轴表示输出是 0 或 1：

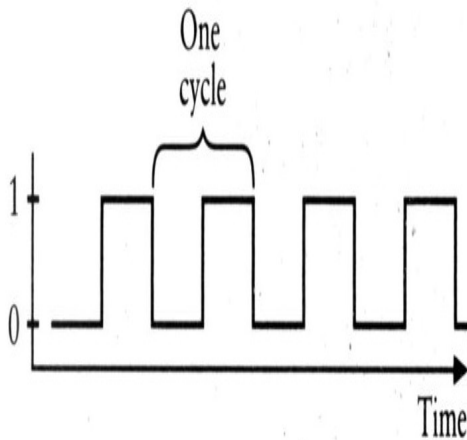


时间

此图表示随着时间的变化，振荡器的输出在 0 和 1 之间交替变化。基于这个原因，振荡器有时称为 时钟（clock），因为通过对振荡次数记数还可确定时间。

那么，振荡器运行的速度有多快呢？也就是说，金属簧片上下跳动的频率是多少？每秒有多少次呢？很明显，这依赖于继电器是如何构造的。容易想到，一个大的、笨重的继电器只能迟钝地上下摆动；而一个小的、轻巧的继电器可以迅速地跳动。

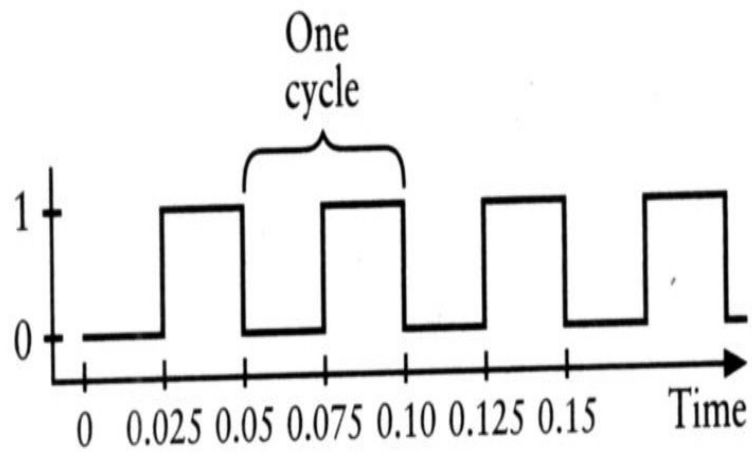
我们把振荡器从某个时间的输出开始，经历一段变化又回到同样输出的这一段间隔称为 振荡器的一个循环（cycle）：



一个循环

时间

一个循环所需要的时间称为振荡器的周期。假设一个振荡器的周期是 0.05秒, 则可以在水 平轴上标出时间:



一个循环

0 0.025 0.05 0.075 0.10 0.125 0.15 时间

振荡器的频率是周期的倒数。本例中，若振荡器的周期是 0.05 秒，则其频率是 $1 \div 0.05$ 秒，即每秒钟 20 个循环。这表明振荡器的输出每秒钟改变 20 次。

每秒循环数与每小时英里数、每平方英寸磅数、每份食物（饮料）的卡路里数等毋需多解释的术语一样是一个很容易理解的概念，但已不常用。为了纪念第一个发送和接收无线电波的人——鲁道夫·赫兹 (1857-1894)，我们用“赫兹”这个词表示每秒的循环数。这个用法

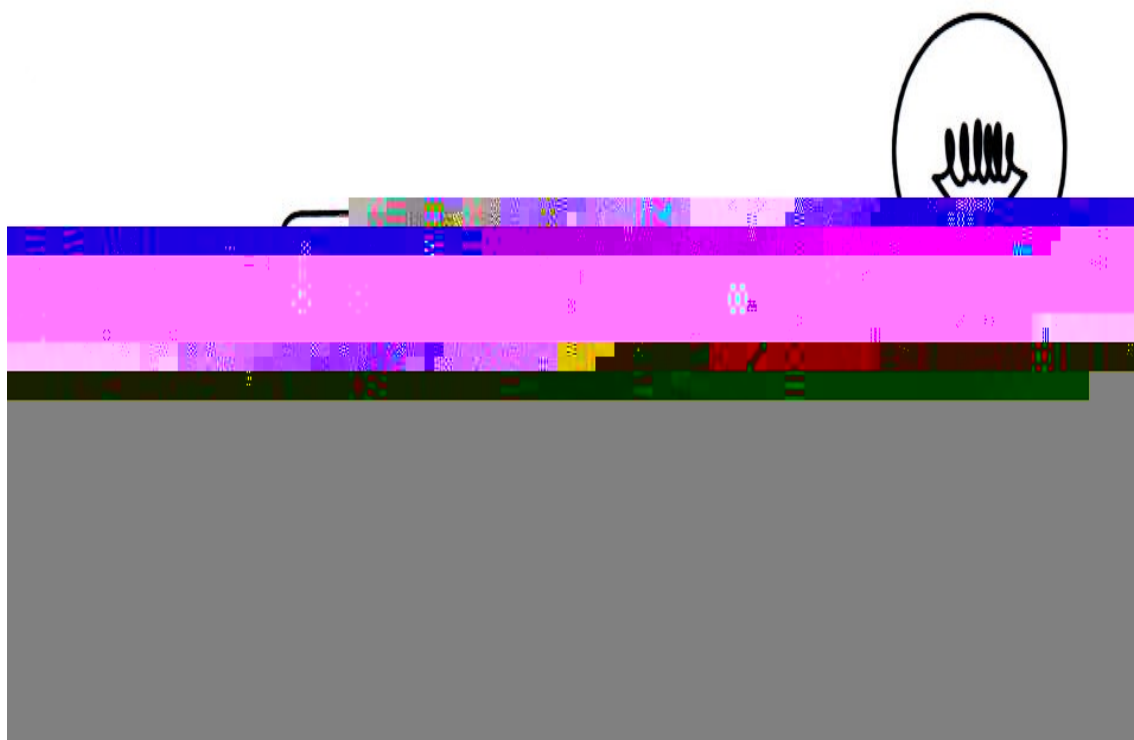
始于20世纪20年代的德国，后来传到其他国家。于是，我们可以说这个振荡器的频率是 20赫兹，或直接简写为 20Hz。

到目前为止，我们只是在假设一个振荡器的速度。到本章末尾，我们可以构造一种器件 来真正地测量一个振荡器的速度。

为了构造这个器件，先看一个用特殊方式连接的一对或非门。或非门的特点是只有两个 输入都为 0时，输出才为 1：

NOR	0	1
0	1	0
1	0	0

下图是含有两个或非门、两个开关和一个灯泡的电路：



注意图中奇特的连接方式：左边或非门的输出是右边或非门的输入，右边或非门的输出是左边或非门的输入。这是一种反馈。事实上，这和在振荡器中类似，输出又返回作为一种输入。这是本章中大部分电路的特点。



在上图电路中，一开始，只有左边或非门的输出有电流，因为它的两个输入均为 0。现在 闭合上面的开关，左边或非门的输出变为 0，于是右边或非门的输出变为 1，灯泡点亮：

神奇之处在于当你断开上面的开关时，由于或非门的输入中只要有一个为 1，其输出就是



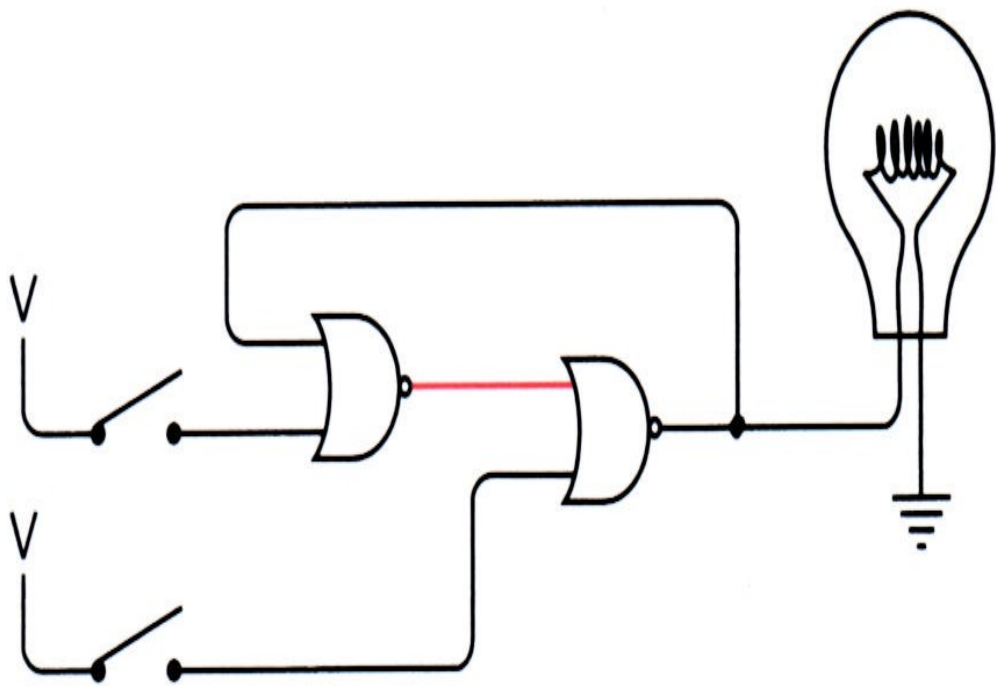
0，因而左边或非门的输出不变，灯泡仍然亮着：

你不觉得奇怪吗？两个开关都断开着，和第一幅图一样，但灯泡却亮着。这种情形和以前所见到的完全不同。通常，一个电路的输出仅仅依赖于输入，这里的情况却不一样。无论断开或闭合上面的开关，灯泡总是亮着。这里开关对电路没有什么影响，原因是左边或非门的输出一直是 0。



现在闭合下面的开关。由于右边或非门的输入中有一个是 1，则其输出变为 0，灯泡熄灭。左边或非门的输出此刻变为 1：

现在，再断开下面的开关，灯泡仍旧不亮：



此电路和初始电路一样。然而这回却是下面开关的状态对灯泡没有什么影响。总结起来就是：

- 闭合上面的开关使灯泡点亮，当再断开时，灯泡仍然亮着。

- 闭合下面的开关使灯泡熄灭，当再断开时，灯泡仍然不亮。电路的奇特之处是：有时当两个开关都断开时，灯泡亮着；而有时，当两个开关都断开

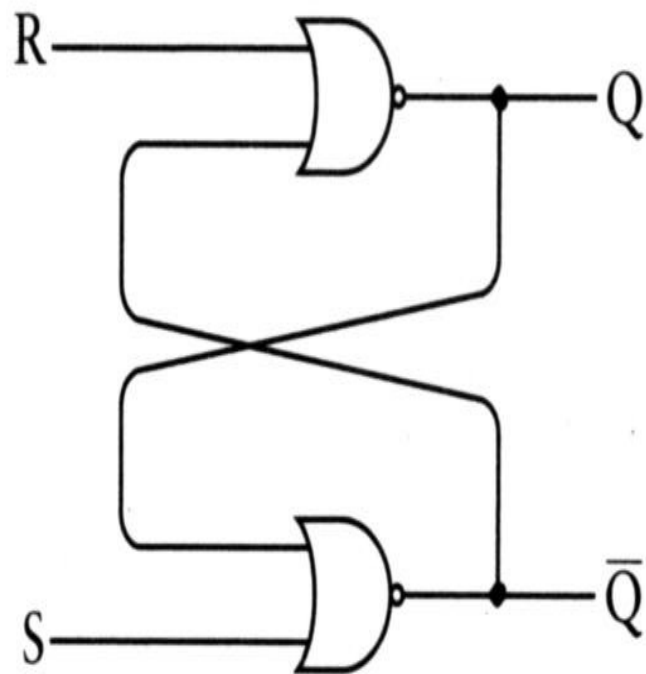
时，灯泡却不亮。当两个开关都断开时，电路有两个稳定状态，这样的电路称为触发器。触发器是 1918年在英国射电物理学家 William Henry Eccles(1875-1966)和F.W.Jordan的工作中发明的。

触发器电路可以保持信息，换句话说，它有记忆性。它可以“记住”最近一次是哪个开关先闭合的。如果你遇到这样一个触发器，它的灯泡亮着时，你可以确定最近闭合的是上面的开关；而灯泡灭着时则是下面的开关。

触发器和跷跷板很像。跷跷板有两个稳定状态，它不会长期停留在不稳定的中间位置。你只要一看跷跷板就知道哪边是最近被压下来的。

触发器是十分关键的工具，尽管你现在可能还没看出来。它们赋予电路“记忆”，使其知道以前曾有过的状态。想像一下，如果你没有记忆力，你该如何去数数，你记不住你刚数过的数，当然也无法确定下一个数是什么。同样，一个能计数的电路（本章后面要提到）必定需要触发器。

触发器有很多种，刚才所看到的是最简单的一种，称为 R-S（或 Reset-Set，复位/置位）触发器。下面以对称的方式把它重新绘出来：



用于点亮灯泡的输出称为 Q ，另一个输出 \bar{Q} 是 Q 的倒置。如果 Q 是 0， \bar{Q} 就是 1，反之亦然。两个输入端 S (Set) 和 R (Reset) 分别表示置位和复位。你可以把“置位”理解为把 Q 设为 1，而“复位”是把 Q

设为0。当S为1时（对应于前面图中闭合上面开关的情况），Q变为1而Q-变为0；当R为1时（对应于前面图中闭合下面开关的情况），Q变为0而-Q变为1。当S和R都为0时，输出保持Q原来的状态。输入与输出的关系小结于下表中：

Inputs		Outputs	
S	R	Q	\bar{Q}
1	0	1	0
0	1	0	1
0	0	Q	\bar{Q}
1	1	Disallowed	

输入 输出

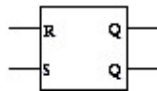
禁止

这张表称为功能表、逻辑表或真值表。它指明不同的输入组合能产生不同的输出结果。由于

R-S触发器有两个输入端，因而不同的输入组合有4种，分别对应于表中的4行。注意表中倒数第2行中S和R均为零，而输出标识为Q和-Q。这表示当S和R输入均为零时，

Q和-Q端的输出保持S、R同时设为0以前的输出值。表中最后一行说明S和R输入都为1是非法的、禁止的。这是因为S、R同时为1时，两个输出Q和-Q均为零，这与Q和-Q互为倒置的关系相矛盾。所以，当你用R-S触发器设计电路时，要避免使R、S输入同时为1的情况。

R-S触发器通常画成有两个输入，两个输出的方块图，如下图所示：



R-S触发器能够记住哪一个输入端最近被输入高电位，这确实很有趣。但更有用的电路应该能记住某个特定时间点上上一个信号是0还是1。

在实际构造这种电路之前，先来思考一下它的行为功能。它需要两个输入，其中一个称

为数据端（Data）。像所有数字信号一样，数据端输入可以是0或1。另一个输入称为保持位

（Hold that bit）。通常情况下，保持位设为0，这时，数据端对电路没什么影响。当保持位置为1时，电路就反映出数据端的值。接着，保持位又置为0，这时，电路将记住数据端输入的最近一个值。数据端信号的任何改变不会对电路再有影响。

换句话说，它的功能表可以这样写：

Inputs		Outputs
Data	Hold That Bit	Q
0	1	0
1	1	1
0	0	Q
1	0	Q

输入 输出

数据端 保持位

在前两种情况下，保持位置为 1，Q端输出和数据端输入相同；后两种情况下，当保持位置为0时，Q端输出和它以前的值相同，即保持原状态。注意，后两种情况中当保持位为 0时，Q端输出不再受数据端输入的影响，功能表可以简化表示为：

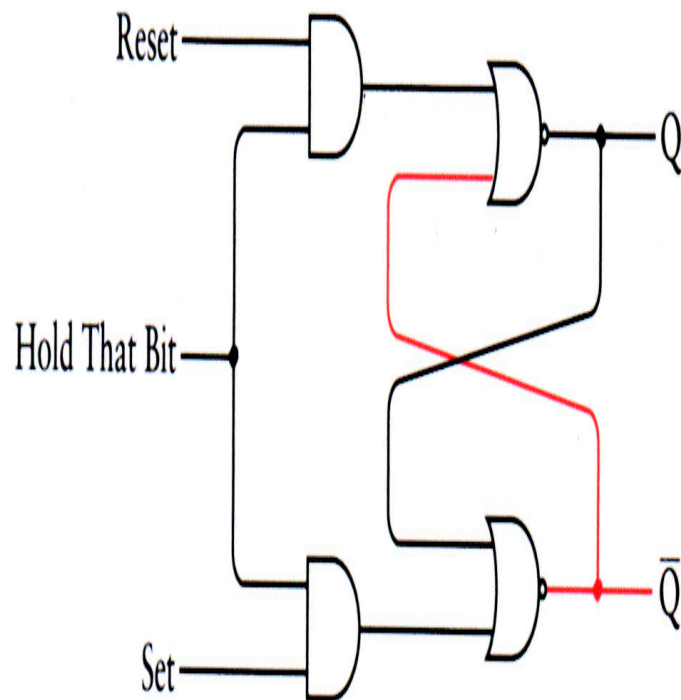
Inputs		Outputs
Data	Hold That Bit	Q
0	1	0
1	1	1
X	0	Q

输入 输出

数据端 保持位

X表示不关心其取值情况，它的值对于电路输出没有影响。

基于**R-S** 触发器来实现保持位的功能要求在输入端增加两个与门，如下图所示：

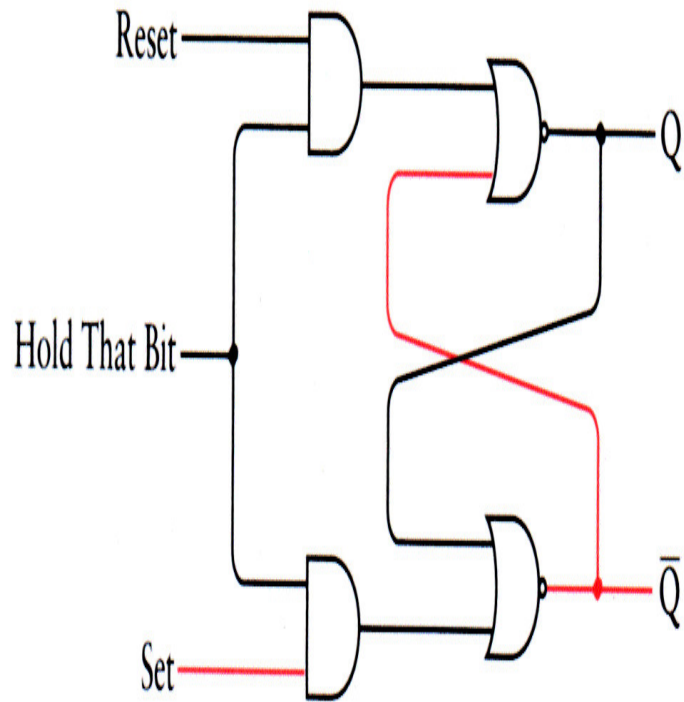


复位

保持位

置位

要使与门输出为 1，两个输入端必须同时为 1。在上图中，Q 输出为 0，而 \bar{Q} 输出为 1。只要保持位置为 0，置位信号对于输出就没有影响：

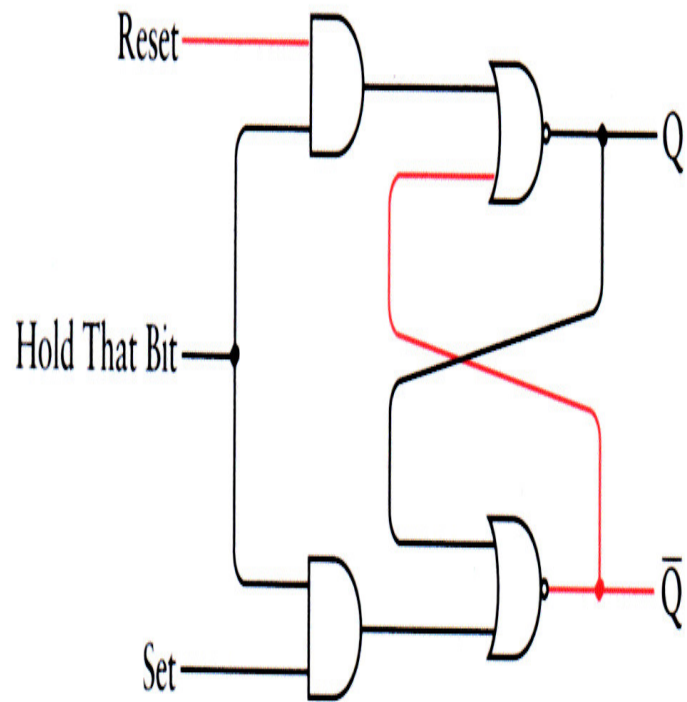


复位

保持位

置位

同样，复位信号对电路输出也没有影响：

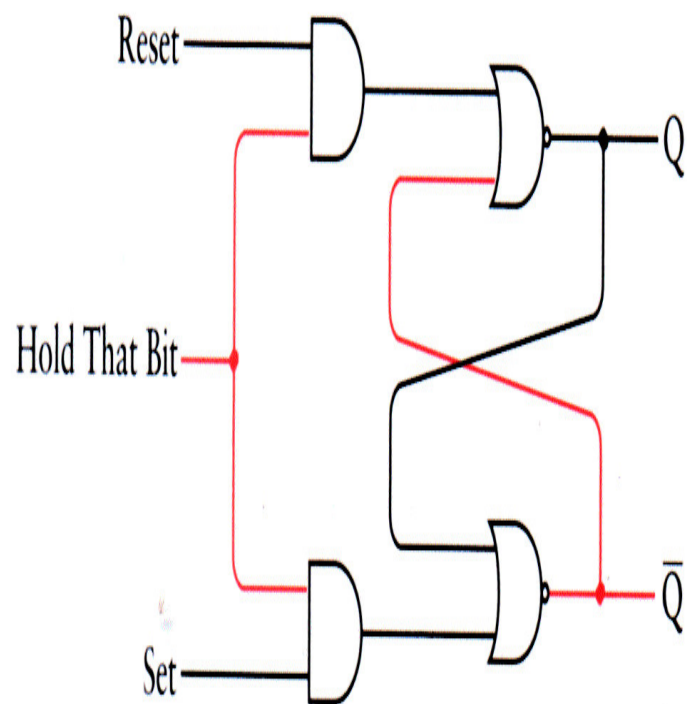


复位

保持位

置位

只有当保持位信号是 1 时，电路的功能才和前述的 R-S 触发器相同：



复位

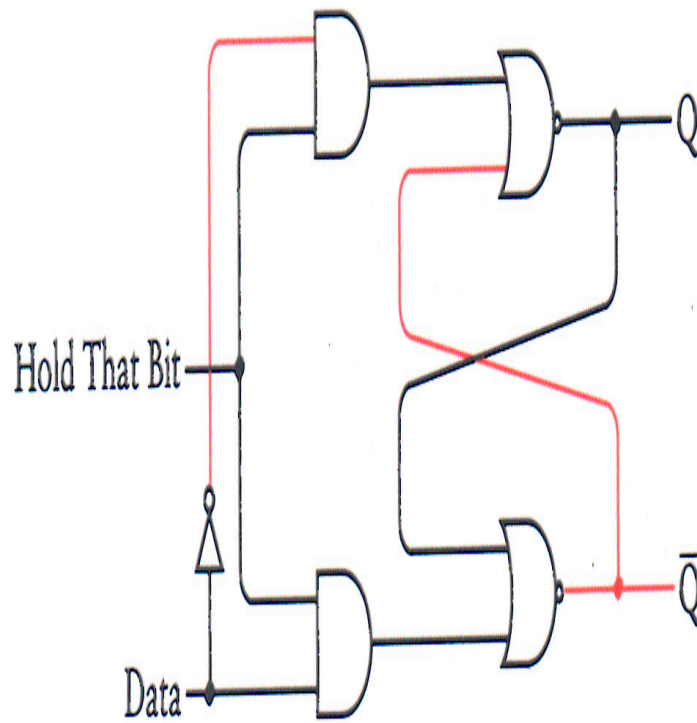
保持位

置位

这时，由于上面与门的输出和复位端输入相同，而下面与门的输出和置位端输入相同，所以此电路的功能就和普通的 **R-S** 触发器是一样的了。

但我们还没有达到目标，我们只想要两个输入，而不是三个，怎么办呢？前面讲过 **R-S** 触发器中两个输入同时为 1 的情况是禁止的；而两个输入同时为零的情况没有什么意义，因为那只是输出保持不变的简单情况。这里，只要将保持位置为 0，就可以完成同样的功能。

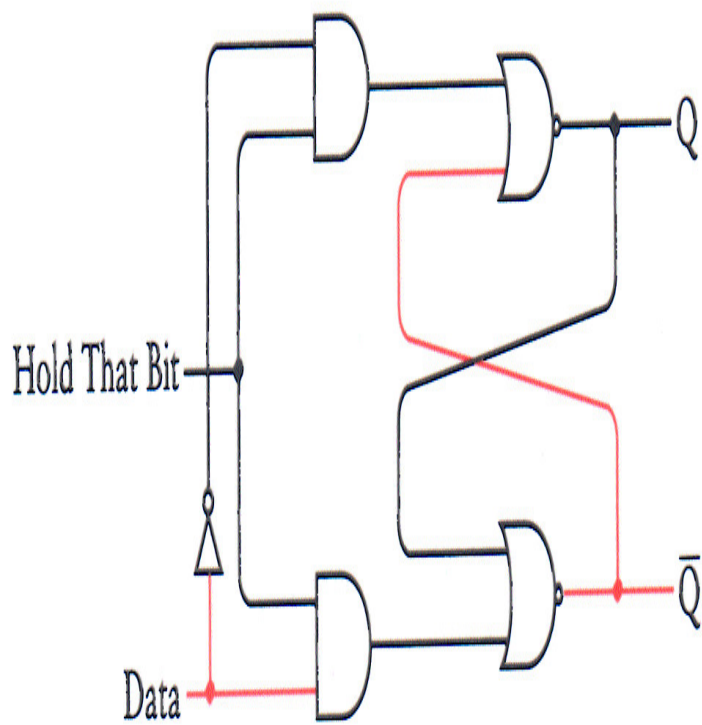
可见，真正有意义的输入是 **S** 为 0，**R** 为 1 或 **R** 为 0，**S** 为 1。把数据端信号当作置位信号，它取反后的值就是复位端信号，如下图示：



保持位

数据端

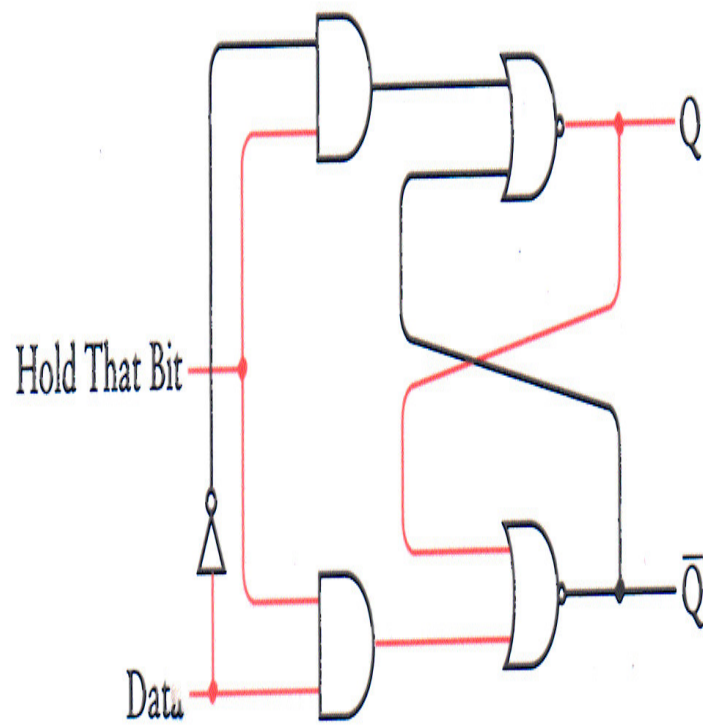
在这种情况下，S和R输入以及输出Q均为0，Q-为1。只要保持位为0，数据端输入对于电路输出就没有影响：



保持位

数据端

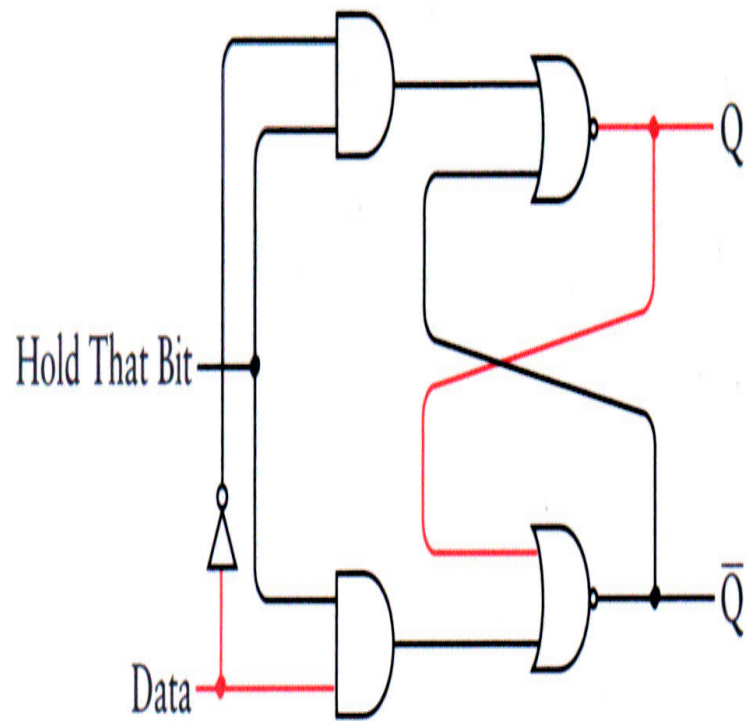
当保持位为 1 时，电路反映出数据端输入的值：



保持位

数据端

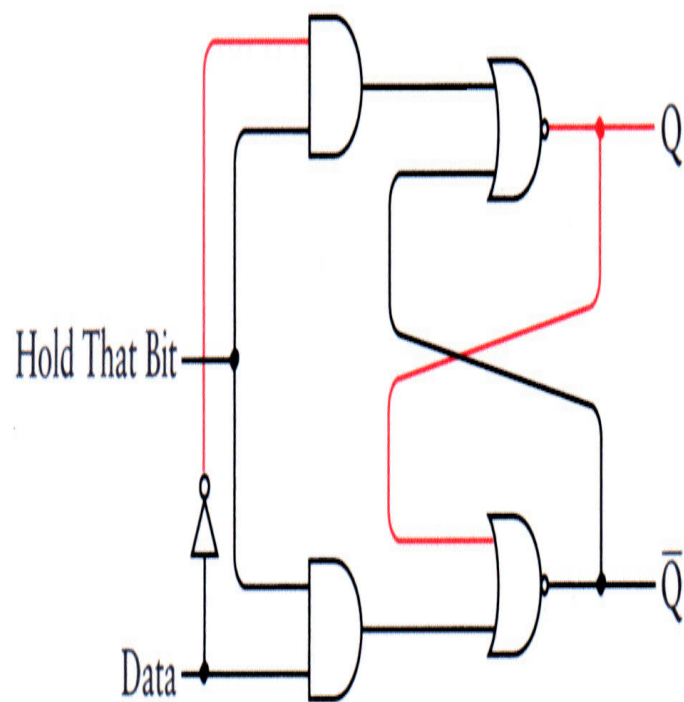
Q端输出现在和数据端输入是一致的， \bar{Q} -则相反。现在，保持位又回到0:



保持位

数据端

这时，电路会记得当保持位最后一次置为 1 时数据端输入的值。数据端以后的变化对电路 的输出没有影响：



保持位

这个电路称为电平触发的 D 型触发器，D (Data) 表示数据端输入。所谓电平触发 是指当 保持位输入为某一特定电平（本例中为“1”）时，触发器才对数据端的输入值进行保存。（很快，你将会看到另一种形式的触发器。）

通常情况下，当这样一个电路出现在书中时，输入并不被标为保持位，而是标为“时钟”。当然，这个信号并不是一个真的时钟，但它有时却具有类似钟一样的属性，即在 0 和 1 之间是有规律地来回变化。但是现在时钟只是用来指示什么时候保存数据：

时钟

数据端

把数据端简写为 D，时钟端简写为 Clk，其功能表如下所示：

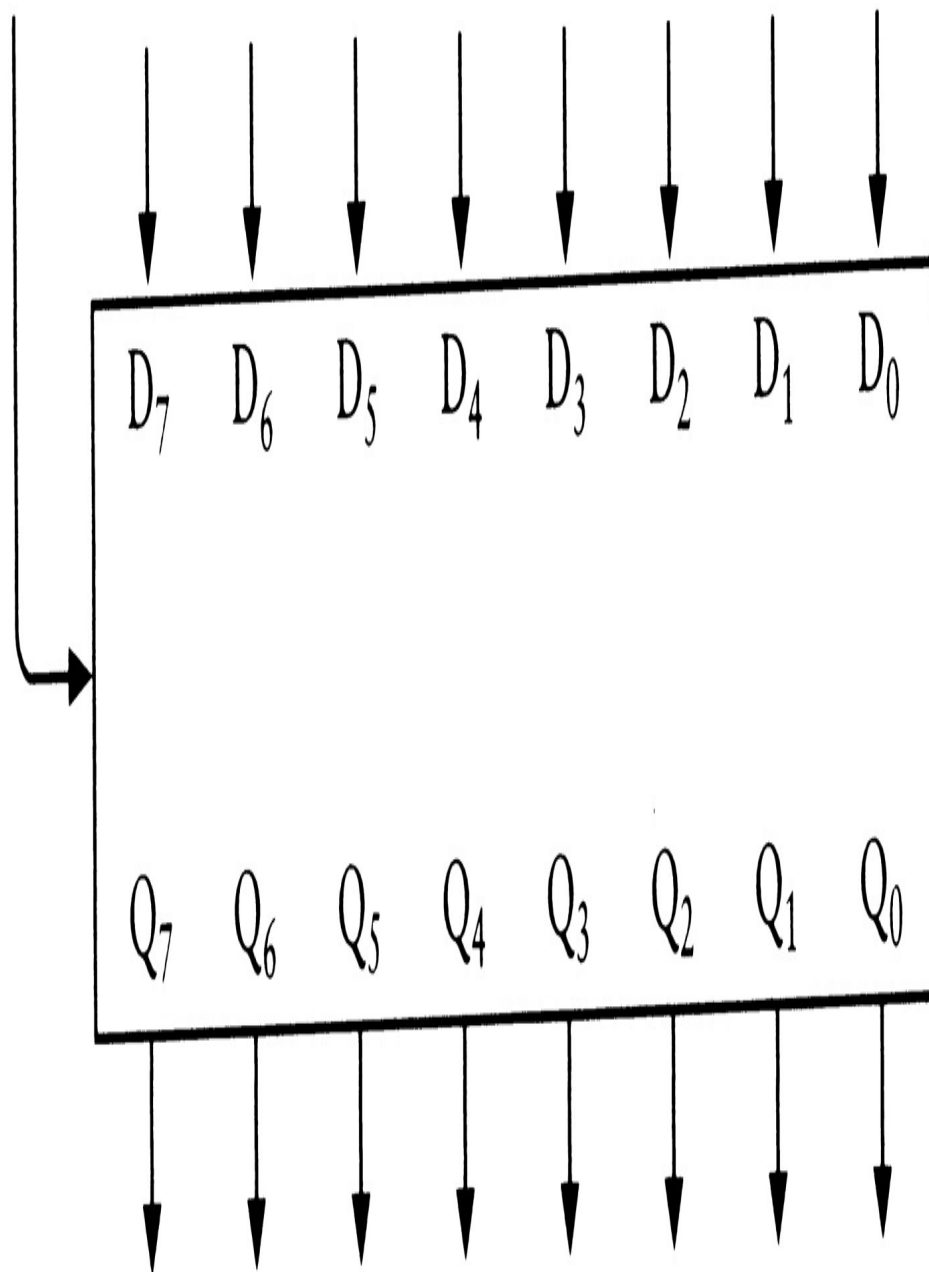
Inputs		Outputs
D	Clk	Q
0	1	0
1	1	1
X	0	\overline{Q}

输入 输出

这个电路就是所谓的电平触发的 **D**型锁存器，它表示电路锁存住一位数据并保持至将来使用。它也可以称为 **1**位存储器。本书将在第 **16**章中说明如何将多个 **1**位存储器连起来以构成多位存储器。

在锁存器中保存多位值是很有用的。假如你想用第 **12**章中的加法机把三个 **8**位数加起来，你可以在第 **1**行开关上输入第一个加数，在第 **2**行开关上输入第二个加数，但是你必须把第一次加法运算的结果记录下来，然后以同样方式把记下来的结果和第三个加数再用开关输入。这是十分麻烦的。

使用锁存器可以解决这个问题。让我们把 **8**个锁存器集成到一个盒子里，形成一个 **8**位锁存器。每个锁存器用到两个或非门、两个与门和 **1**个反向器。时钟端输入是互相连在一起的。结果如下图所示：

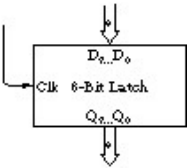


这个锁存器一次可以保存 8 位数。上面的 8 个输入标为 $D_7 \sim D_0$ ，下面的 8 个输出标为 $Q_7 \sim Q_0$ 。左

0 7 0 7

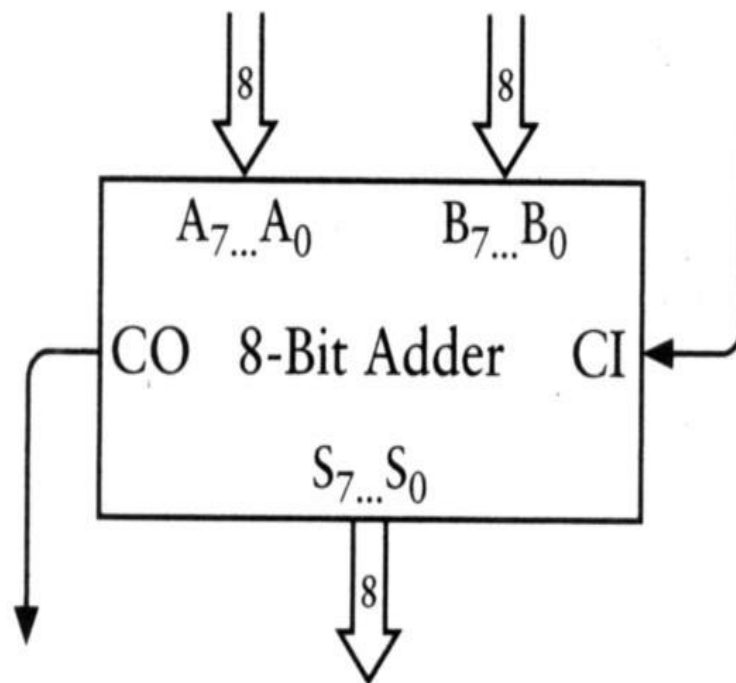
边的输入是时钟（ Clk ），时钟信号通常为 0。当时钟信号为 1 时， D 端输入被送到 Q 端输出。当

时钟信号变为 0 时，8 位输出值保持不变，直到时钟信号再次被置为 1。8 位锁存器也可以画成下面的样子：



8 位锁存器

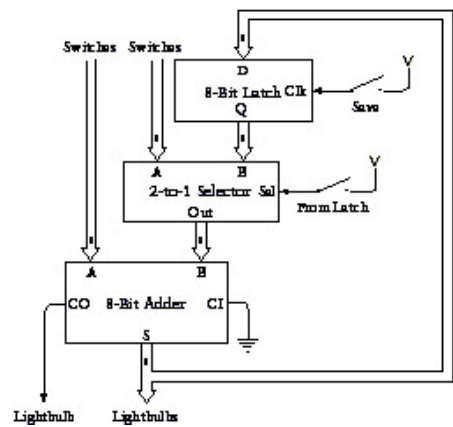
下面是8位加法器：



8 位加法器

通常（先不考虑上一章的减法），8个A输入和8个B输入是连在开关上的，CI（进位输入）端接地，8个S（和输出）和CO（进位输出）端连着灯泡。

经修改，8位加法器的输出既与灯泡相连，也作为8位锁存器的数据端 (D) 输入。标为“保存”（Save）的开关是锁存器的时钟输入，用于保存加法器的运算结果：



开关 开关

8位锁存器

保存

A B

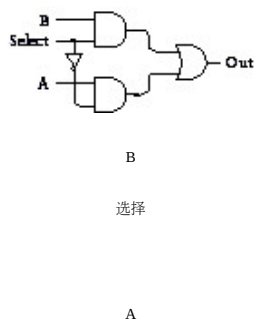
2-1 选择器 输出

来自锁存器

8位加法器

灯泡 灯泡

标识为 2-1选择器的方块是让你用一个开关来选择加法器的 **B**端输入是取自第 2排开关还是 取自锁存器的 **Q**端输出。当选择开关闭合时，就选择了用 8位锁存器的输出作为 **B**端输入。 2-1 选择器用了 8个如下电路：



如果选择（ **Select**）端输入为 1，或门的输出和 **B**端输入是一样的。这是因为上面与门的输出和**B**端输入是一样的，而下面与门的输出是 0。同样，如果选择端输入是 0，或门的输出则和 **A**端输入是一样的。总结起来如下表所示：

Inputs			Outputs
Select	A	B	Q
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

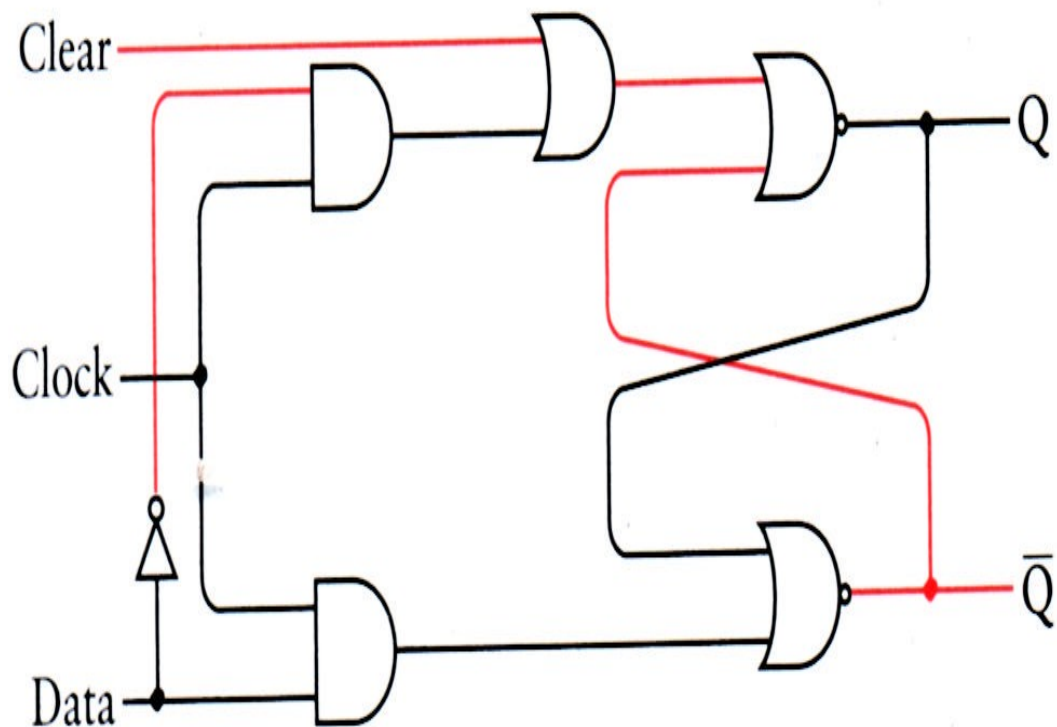
输入 输出

输出端 A B

修改后的加法机中包含了 8 个这样的 1 位选择器。所有选择端的信号输入是连在一起的。改进过的加法机不能很好地处理进位输出 (CO) 信号。如果两个数的相加使进位输出信号

为 1，则当下一个数再加进来时，这个信号就被忽略了。一个可能的解决方法是使加法器、锁存器、选择器均为 16 位宽度，或者至少比你可能遇到的最大的和的位数多一位。这个问题会在第 17 章中专门讲述。

对加法机一个更好的改进方法是完全去掉一排开关，但是这需要先对 D 触发器做一点儿小的改进，对它加一个或门和一个称为清零 (Clear) 的输入信号。清零信号通常为 0，但当它为 1 时，Q 输出为 0，如下图所示：



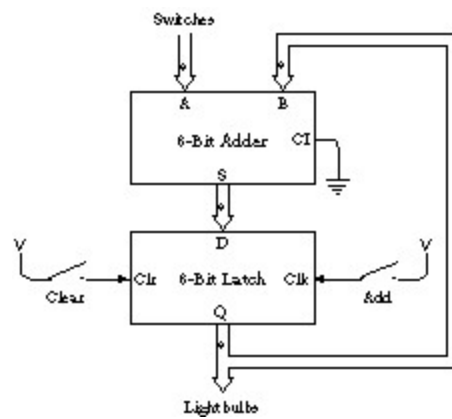
时钟

数据端

无论其他信号是什么，清零信号总迫使 Q 输出为 0，起到了给触发器清零的作用。你也许还不明白为什么要设置这个信号，为什么不能通过把数据端输入置 0 和时钟端输入

置 1 来使触发器清零呢？这也许因为我们并不能控制数据端的输入。下图中，8 个锁存器连着 8

位加法器的输出：



开关

8位加法器

8位锁存器

清零 相加

灯泡

注意，标识为“相加” (Add)的开关此刻控制着锁存器的时钟输入。你可能会发现这个加法器比前面那个好用，尤其是当你需要加上一长串数字时。刚开始

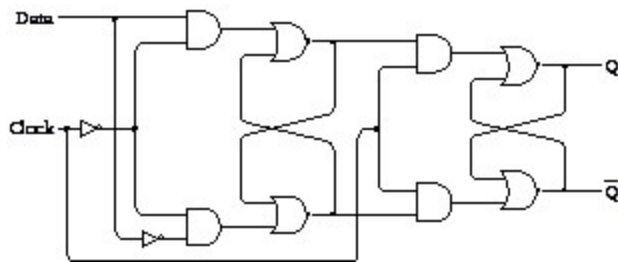
时，按下清零开关，这个操作使锁存器输出为 0，并熄灭了所有的灯泡，同时使加法器的 B端 输入全为 0。接着，通过开关输入第一个加数，闭合“相加”开关，则此加数反映在灯泡上。再输入第二个数并再次闭合“相加”开关，由开关输入的 8位操作数加到前面的结果上，其和 输出到灯泡。如此反复，可以连加很多数。

触发器是电平触发式的，意思是说只有在时钟端输入从 0变到1后（即高电平时），数据端 输入的值才能保存在锁存器中。注意，在时钟端输入为 1期间 ,数据端输入的任何改变都将反 应在Q或Q-端的输出值上。

对一些应用而言，电平触发时钟输入已经足够用了；但对另外一些应用来说，边沿触发 时钟输入更为适用。对于边沿触发器而

言，只有当时钟从 0 变到 1 的瞬间，输出才会改变。在 电平触发器 中，当时钟输入为 0 时，数据端输入的任何改变都不会影响输出；而在边沿触发器 中，当时钟输入为 1 时，数据端输入的改变也不会影响输出。只有在时钟输入从 0 变到 1 的瞬间，数据端的输入才会影响边沿触发器的输出。

边沿触发的 D 型触发器是由两级 R-S 触发器按如下方式连接而成的：

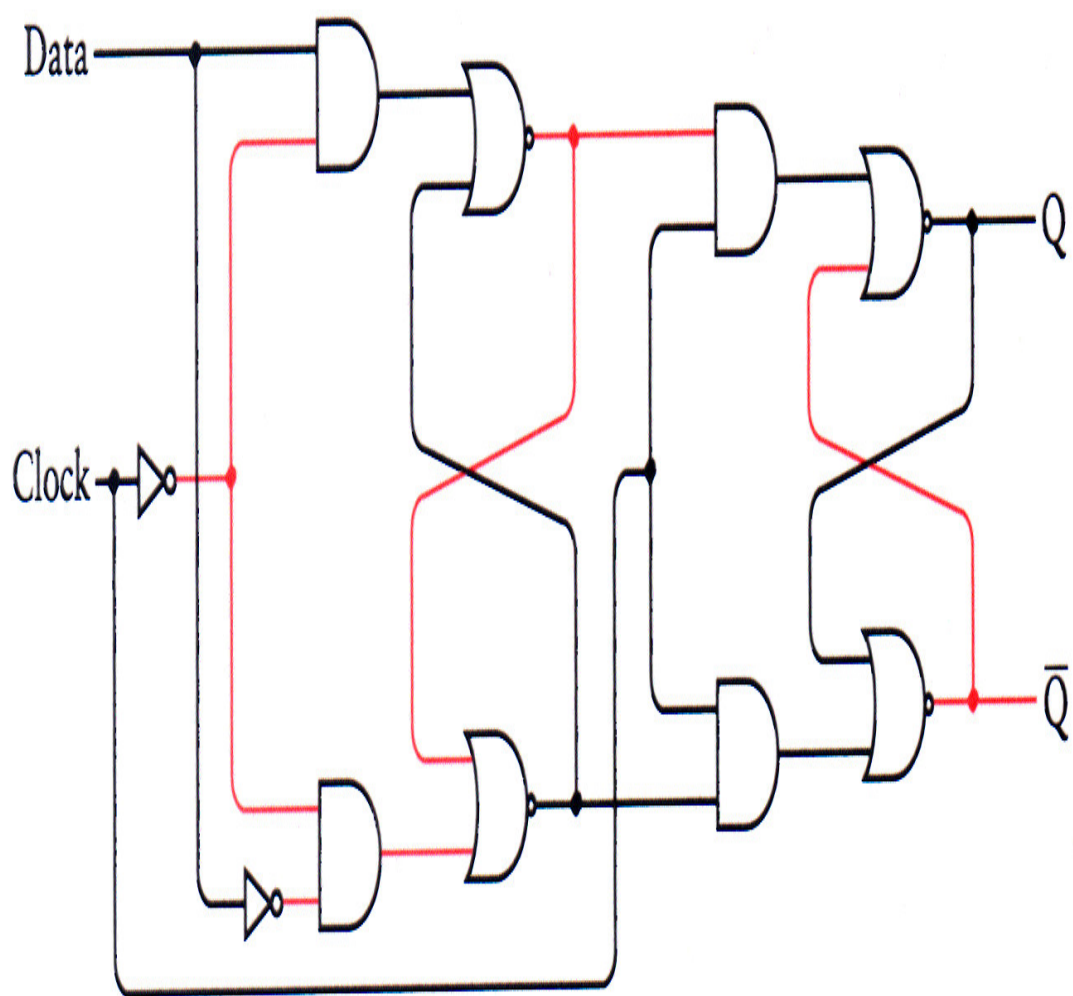


数据端

时钟

这时，时钟输入既控制着第一级，也控制着第二级。但是应该注意到时钟信号在第一级中取了反，这意味着除了当时钟信号为零时保存数据外，第一级工作原理和 D 型触发器完全相同。第二级的输出是第一级的输入，当时钟信号为 1 时，它们被保存。总的结论就是只有当时钟信号从 0 变为 1 时，数据端输入才会保存下来。

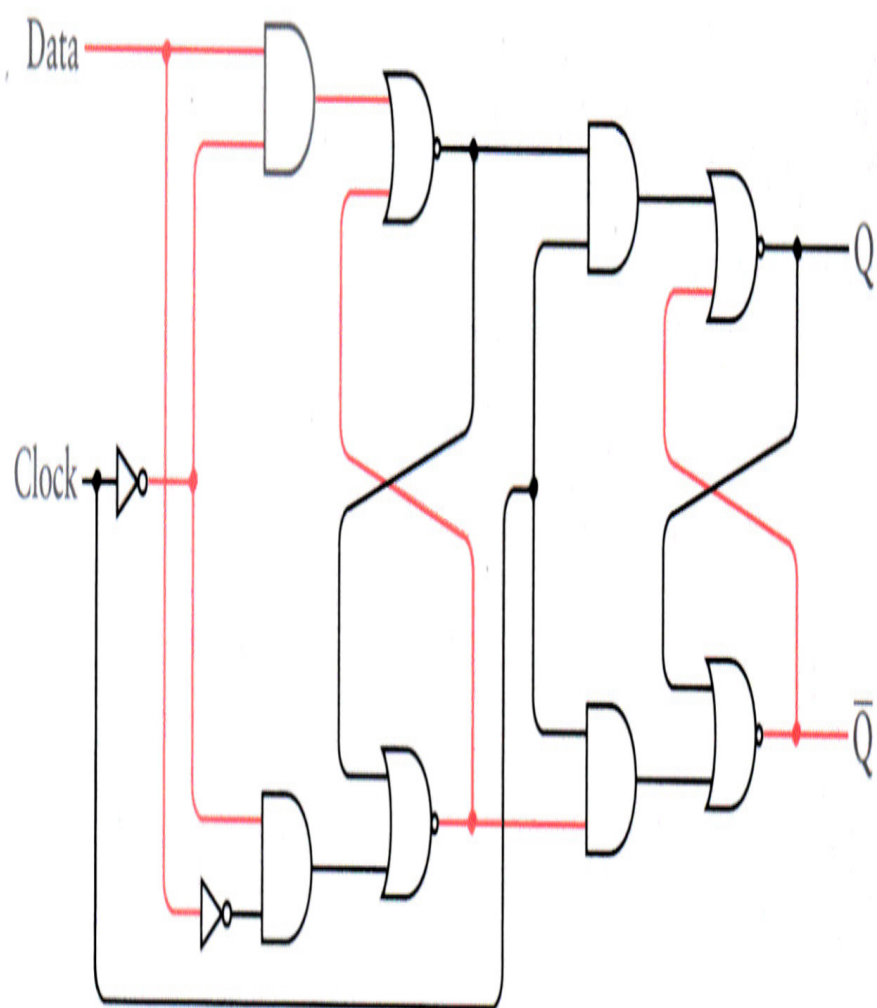
让我们进一步分析。下面是处于非工作状态的触发器，其数据端、时钟输入均为 0，Q 端输出也是 0：



数据端

时钟

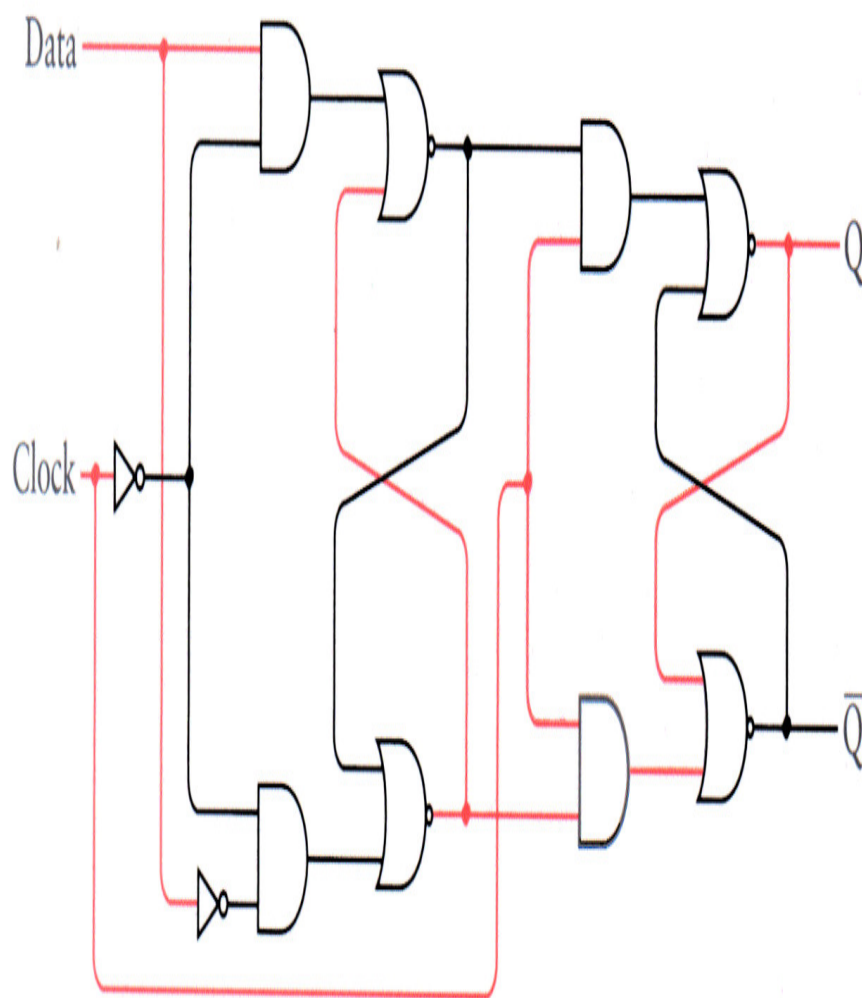
现在，使数据端输入为 1:



数据端

时钟

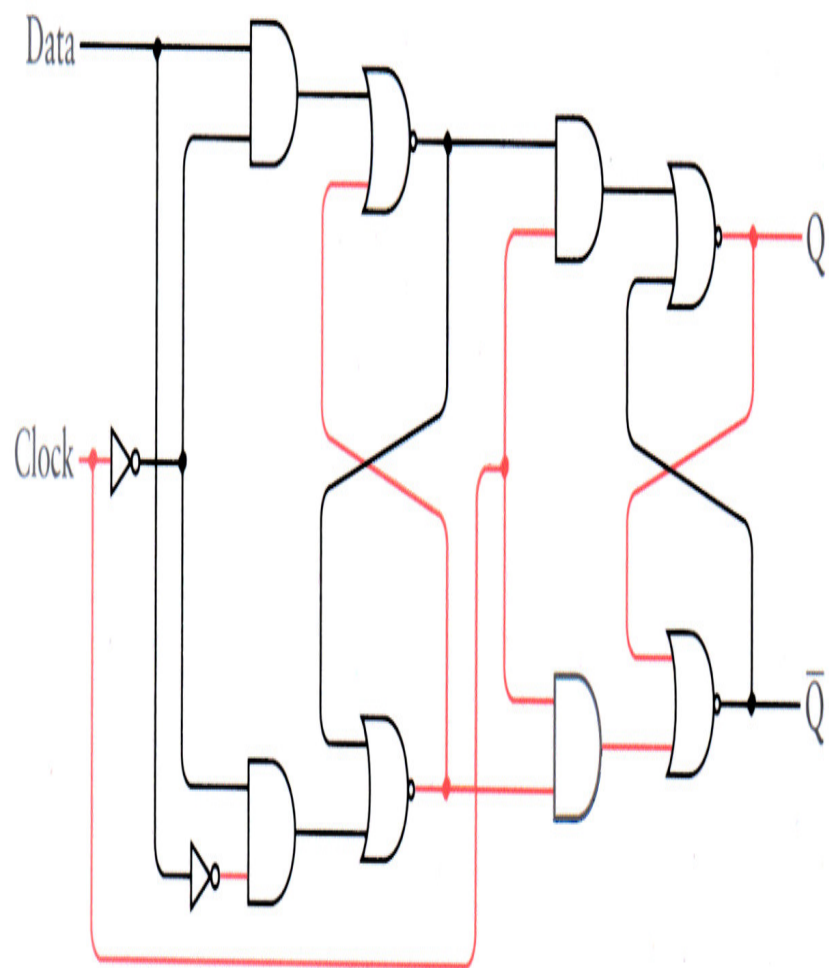
这改变了第一级触发器状态，因为时钟信号取反后为 1。但第二级仍保持不变，因为时钟端输入仍为 0。现在把时钟输入变为 1：



数据端

时钟

这就引起第二级触发器改变，使 Q 端输出变为 1。与前面不同的是现在无论数据端输入如何变化（如变为 0），它也不会影响 Q 端的输出值：



数据端

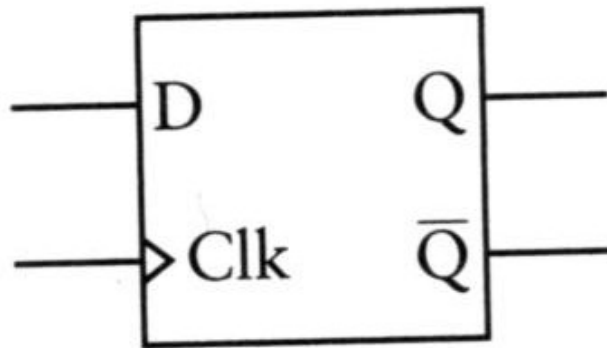
时钟

Q和Q-端输出只有在时钟输入从 0变到1的瞬间才发生改变。

边沿触发的 D型触发器的功能表需要一个新符号来表示这种从 0到1的瞬间变化，即用一个 向上指的箭头（↑）表示：

Inputs		Outputs	
D	Clk	Q	\overline{Q}
0	↑	0	1
1	↑	1	0
X	0	Q	\overline{Q}

箭头表示当 Clk信号从 0变到1时，Q端输出和数据端输入是一样的，这称为 Clk信号的“正跳变”（“负跳变”是从 1到0的转换）。触发器的符号图如下所示：

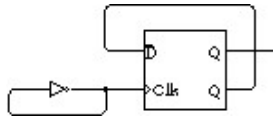


图中的小三角符号表示触发器是边沿触发的。现在向你展示一个使用边沿触发器的电路。先回忆一下本章开始构造的振荡器，振荡器

的输出是在 0和1之间变化的：



把振荡器的输出连到边沿触发的 D型触发器的时钟输入端，并把 Q 端输出连到自己的 D输入端：



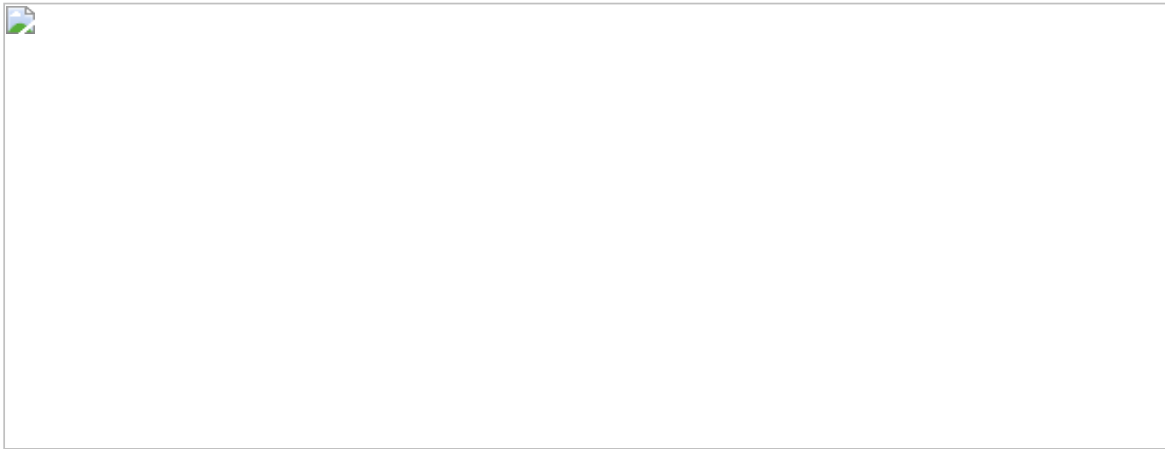
触发器的输出同时又是它自己的输入。（实际上，这种构造可能是有问题的。振荡器是由来回迅速转变状态的继电器构成的。振荡器的输出和构成触发器的继电器相连，而这些继电器不一定能跟上振荡器的速度。为了避免这些问题，这里假设振荡器中继电器的速度比这个电路中其他地方的继电器的速度都慢。）

观察下面的功能表，就可以明白电路中发生的情况了。刚开始时， Clk输入和 Q端输出都是0，则 Q端输出为 1，而它和 D输入是相连的：

Inputs		Outputs	
D	Clk	Q	\bar{Q}
1	0	0	1

输入 输出

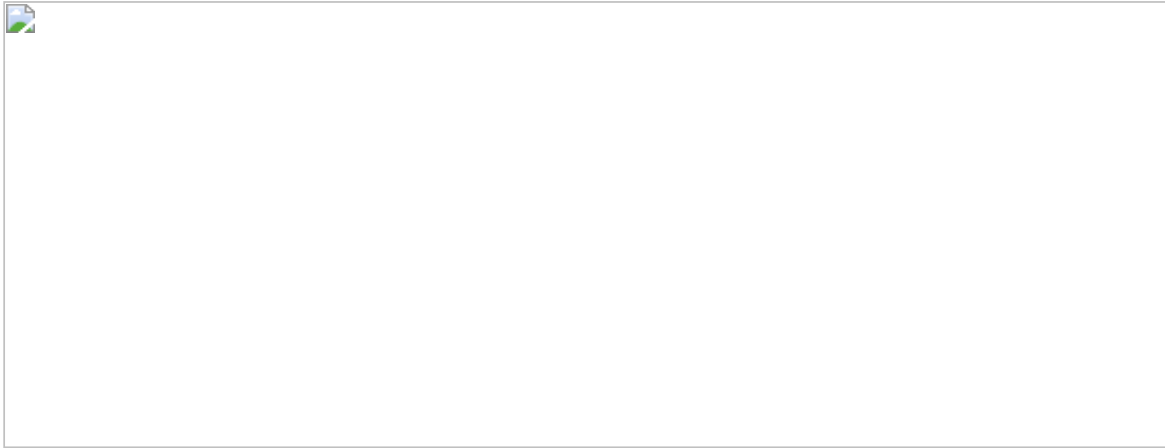
当Clk输入从0变到1后，Q端输出就和 D输入一样了：



输入 输出

但是因为 一端输出变为 0，因而D输入也变为 0。Clk输入现在是 1:

Q



输入 输出

当Clk信号变回为 0时，不会影响输出:



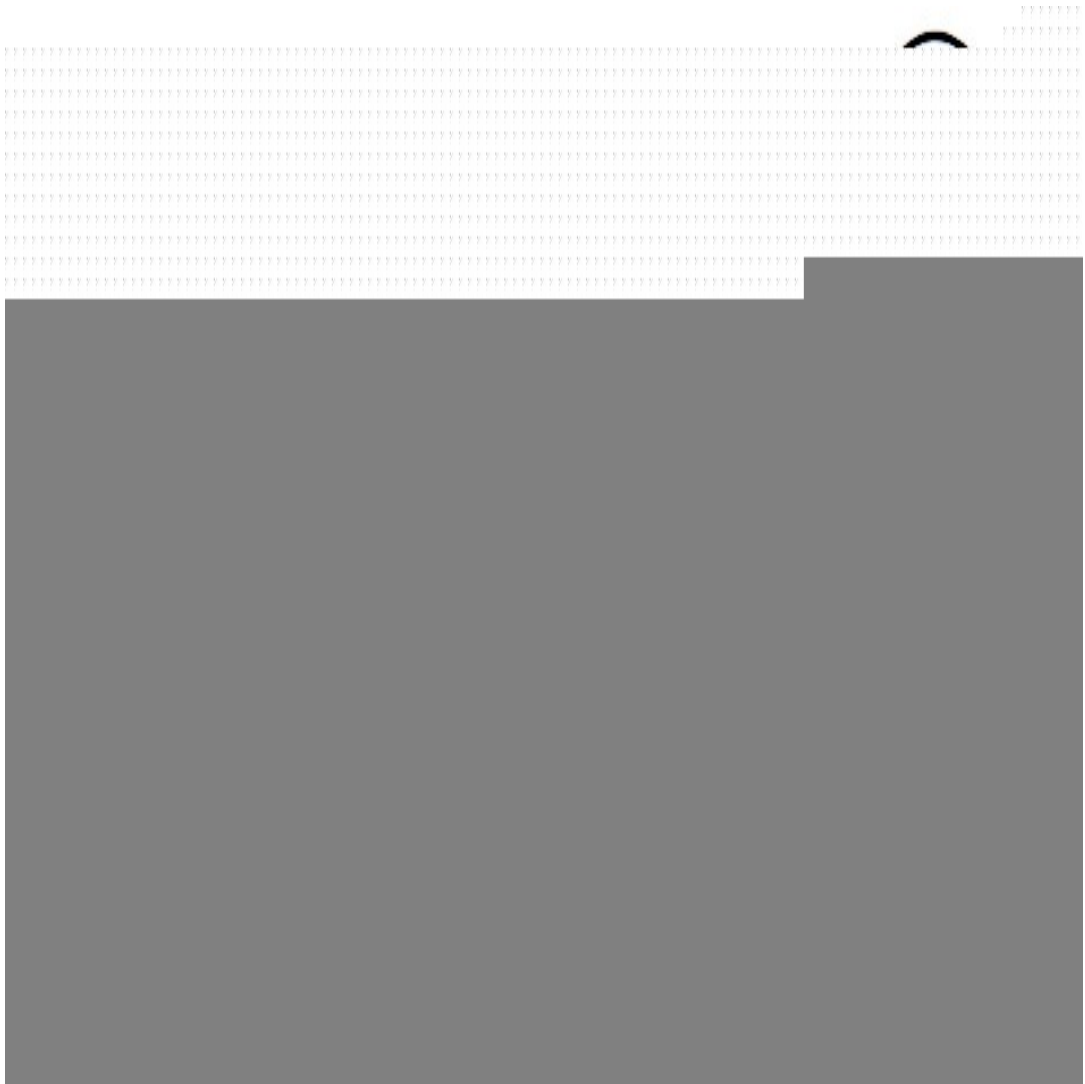
输入 输出

现在Clk信号再变为 1。由于D输入为0，则Q为0且－ 1：
Q为

Inputs		Outputs	
D	Clk	Q	\bar{Q}
1	0	0	1
1	↑	1	0
0	1	1	0
0	0	1	0
0	↑	0	1

输入 输出

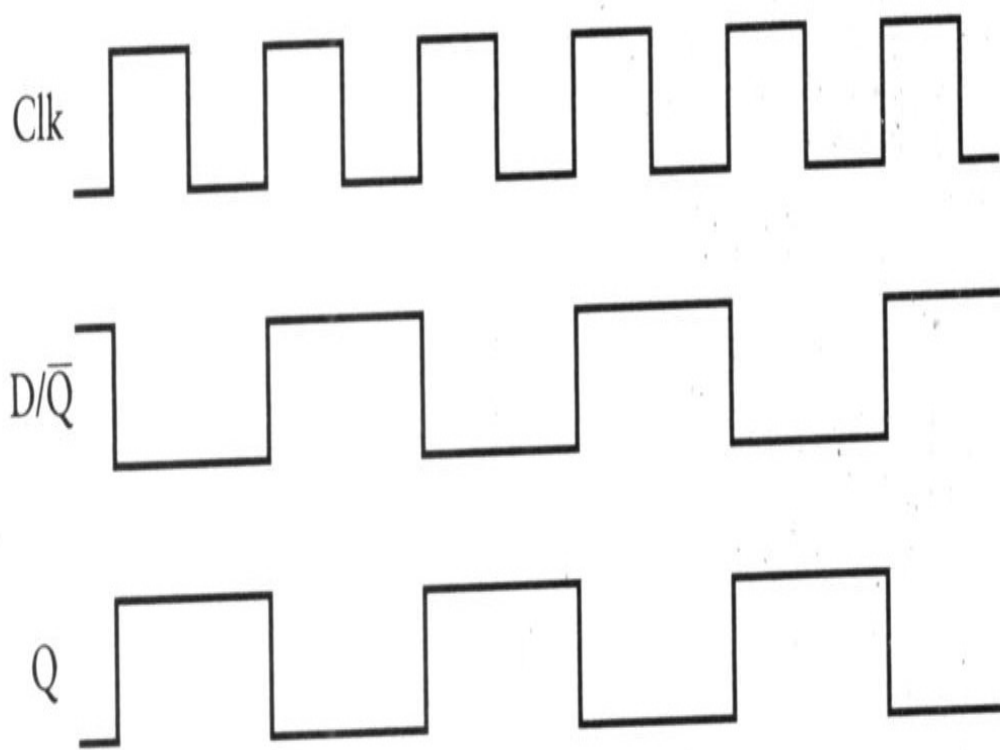
所以D输入也变为 1:



输入 输出

以上发生的情况总结起来就是：每当 Clk输入从0变到1时，Q端输出就发生改变，或者从 0

变到1，或者从 1变到0。看看下面的图，问题就更清楚了：



当Clk输入从 0 变到1时，D的值（与 Q 的值是相同的）被输出到 Q端。当下一次 Clk信号从 0变到1时，同样会改变 D和 Q 的值。

若振荡器的频率是 20赫兹（即每秒 20次循环），则 Q的输出频率是它的一半，即 10赫兹。由于这个原因，这种电路 (其中 Q 输出依循触发器的数据端输入)称为分频器。

当然分频器的输出可以是另一个分频器的 Clk输入，并再一次进行分频。下面是三个分频 器连在一起的情况：

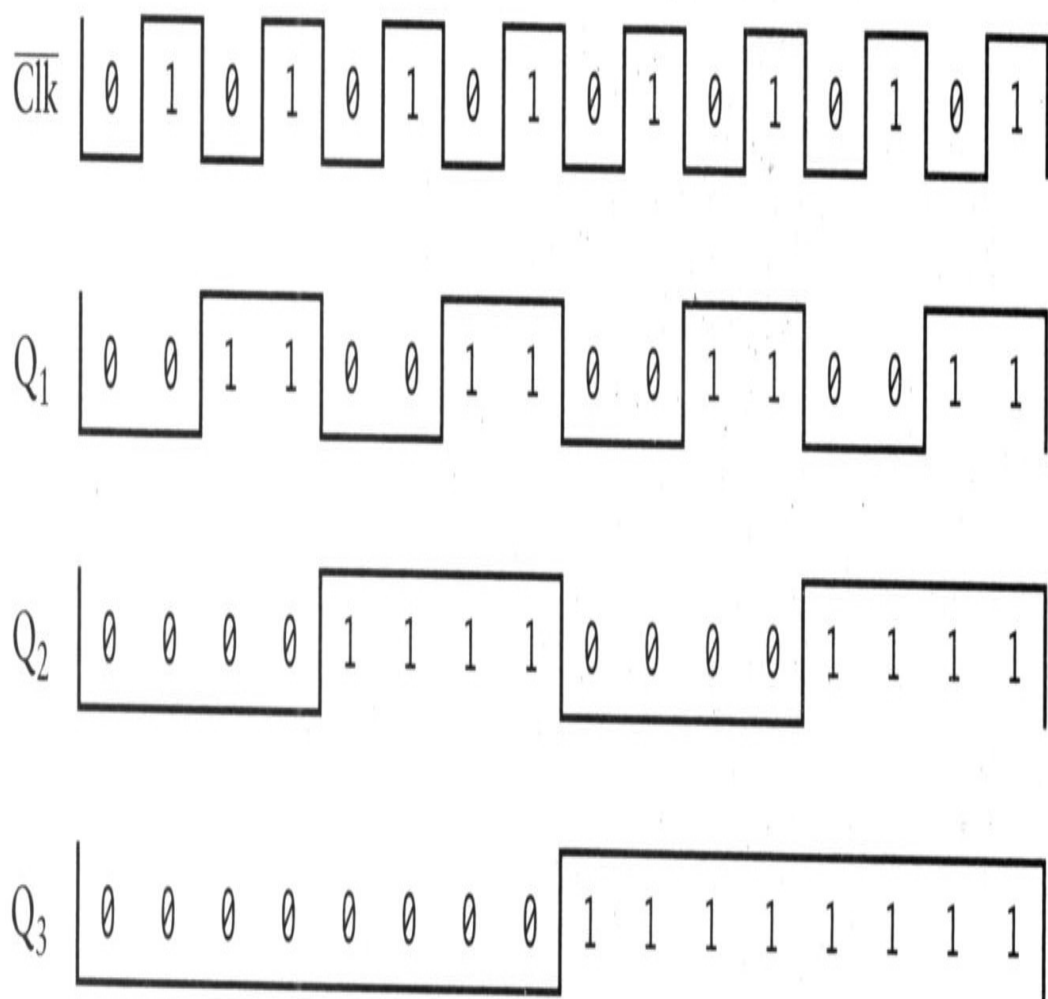


让我们来看一下上图顶部的 4个信号的变化规律：



这里只给出了这幅图的一部分，因为这个电路会周而复始地变化下去。从这个图中，有没有发现使你眼熟的东西？

提示你一下，把这些信号标上 0 和 1：



现在看出来了吗？把这个图顺时针旋转 90 度，读一读横向的 4 位数字，每一组输出都对应了十进制中 0~15 中的一个数：

二进制

十进制

0000

0

0001

1

0010

2

0011

3

0100

4

0101

5

0110

6

0111

7

1000

8

1001

9

1010

10

1011

11

1100

12

1101

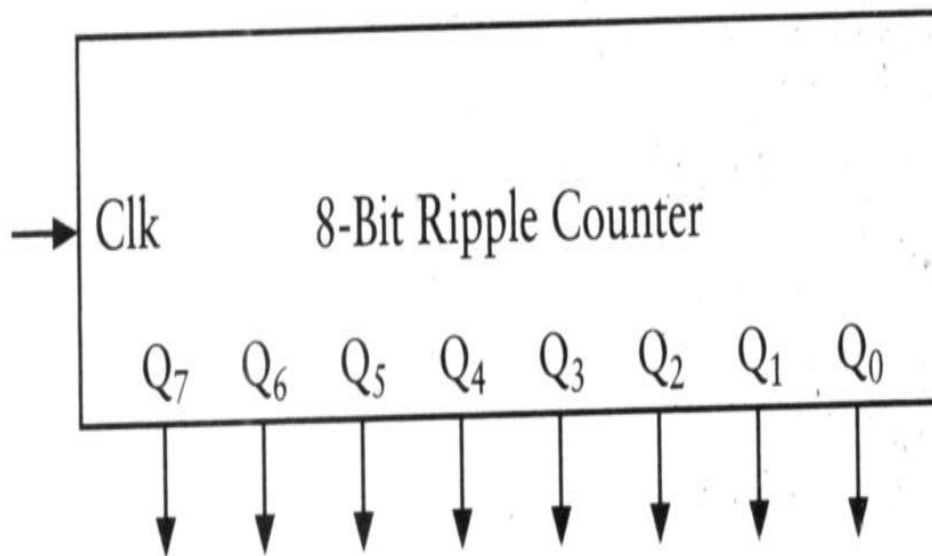
13

1110

14

这个电路只具备了一个计数功能，如果再加上几个触发器，它就可能计更多的数。第 8 章曾指出在一个递增的二进制数序列中，每一列数字在 0 和 1 之间变化的频率是其右边那一列数字变化频率的一半，这个计数器模仿了这一点。时钟信号每一次正跳变时，计数器的输出就递加了 1。

可以把 8 个触发器集成于一个盒子里，构成一个 8 位计数器：



8 位行波计数器

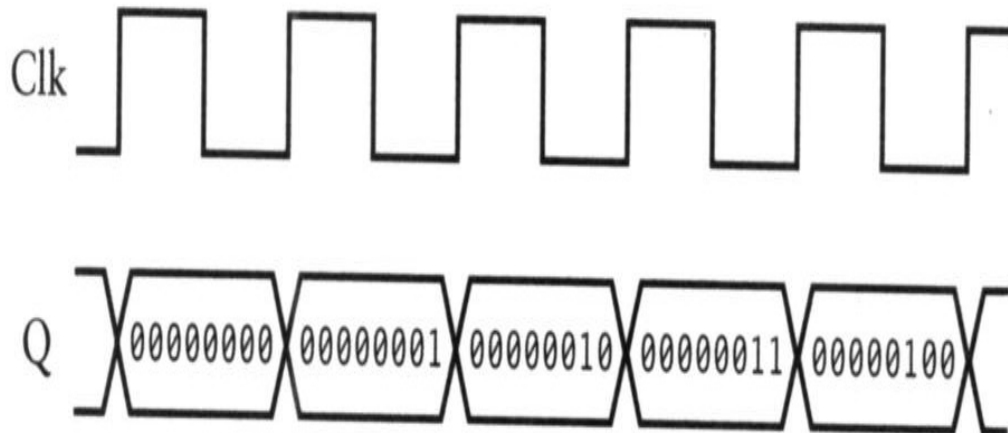
这个计数器称为 8 位行波（异步）计数器，因为每一个触发器的输出都成为下一个触发器的时钟输入。变化是沿着触发器一级一级地传递的，最后一级触发器的变化必定要延迟一些。更复杂的计数器是“并行（同步）计数器”，在这种计数器中，所有输出是同时改变的。

输出端信号已标识为从 $Q_0 \sim Q_7$ ， Q_0 是第一个触发器的输出。如果把灯泡连到这些输出上，

$Q_7 \quad Q_6 \quad Q_5 \quad Q_4 \quad Q_3 \quad Q_2 \quad Q_1 \quad Q_0$

就可以把 8 位结果读出来。

这样一个计数器的时序图可以把 8 个输出分开来表示，也可以把它们一起表示，如下图所示：



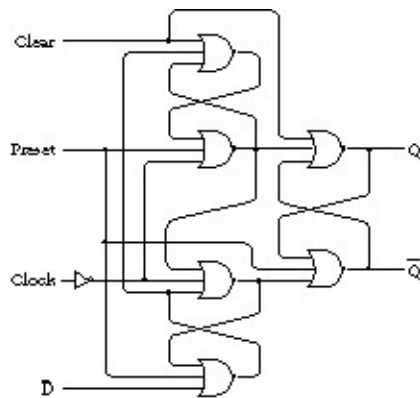
时钟信号的每个正跳变发生时，一些 Q 输出可能改变，另一些可能不改变，但总体上是使

原来的结果递增了 1。本章前面曾提到过可以找到某种方法来确定振荡器的频率，现在这个方法已经找到了。

如果把振荡器连到 8 位计数器的时钟输入上，计数器会显示出振荡器经历了多少次循环。当计

数器总和达到 11111111 时，它又会返回到 00000000。使用计数器确定振荡器频率的最简单方法是把计数器的输出连到 8 个灯泡上。当所有输出为 0 时（即没有一个灯泡点亮），启动一个秒表；当所有灯泡都点亮时，停住秒表。这就是振荡器循环 256 次所需要的时间。假设是 10 秒钟，则振荡器的频率就是 $256 \div 10$ ，或者说是 25.6 赫兹。

当触发器功能增加时，它也变得更复杂。下面这个触发器称为具有预置（Preset）和清零功能的边沿触发的 D 型触发器。



清零

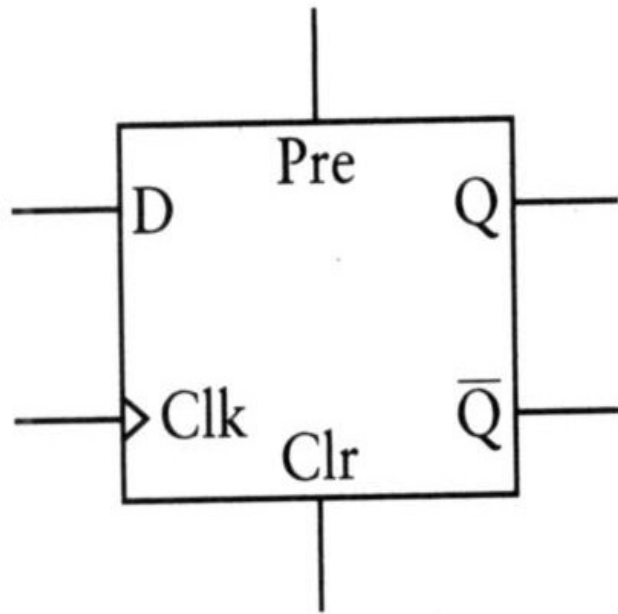
预置

时钟

通常情况下，预置和清零信号输入会忽视时钟和数据端输入，且均为 0。当预置信号输入为1时，Q变为1， \bar{Q} 变为0。当清零信号为 1 时，Q为0， \bar{Q} 变为1（同R-S触发器中的 S和R输入一样，预置和清零信号不能同时为 1）。其他情况下，该触发器的行为和普通边沿触发的 D型触 发器是一样的。

Inputs				Outputs	
Pre	Clr	D	Clk	Q	\bar{Q}
1	0	X	X	1	0
0	1	X	X	0	1
0	0	0	↑	0	1
0	0	1	↑	1	0
0	0	X	0	Q	\bar{Q}

电路图符号可以简化地用下图代替：



现在，我们已经知道如何用继电器来做加法、减法和计数，是不是很有成就感？因为我们所用的硬件是 100 多年以前就存在的东西，我们还有更多的空间去探索。但是先暂时休息一下，不用再去构造什么，回过头来再看看关于数字的问题吧。

第 15 章 字节与十六进制

上一章中的两个改进的加法机清晰地解释了数据路径的概念。在整个电路中，8位值从一个部件传到另一个部件。它们是加法器、锁存器、选择器的输入，经过运算或操作又从这些部件输出。这些数由开关定义，最后由灯泡来表示结果。可以认为电路中的数据路径的宽度是8位。可是，为什么一定是8位，而不是6位、7位、9位或10位呢？

最简单的回答就是这些加法机是在第12章中最原始的加法机上改进而来的，而最原始的那个加法机就是8位。不过，这个解释似乎很缺乏说服力。实际上，用8位的原因是它表示一个字节。

字节这个词大概是在1956年前后由IBM公司最早提出来的。这个词起源于bite，但用y代替了i，以便不会被人误认为它是bit。曾经有一段时期，字节仅仅简单地表示特定数据路径上数据的位数。但是到了20世纪60年代中期，随着IBM的360系统的发展（一种大型复杂的商用计算机），字节这个词专门用来表示8位二进制数。

作为一个8位数，一个字节可以从00000000取值到11111111。这些数可以代表0~255的正数，也可以表示-128~127范围内的正、负数。总之，一个特定的字节可以代表2⁸即256种不同事物中的一个。

8位数事实上是很适用的，字节在很多方面都比一位数优越。IBM采用字节的一个原因就是它们易于以BCD（将在第23章中描述）这种格式保存。巧的是，在以后的各章中你会看到字节用于保存文本也是很合适的，因为世界上大部分书面语言都可以用少于256个字符的

字符集来表示（除了汉语、日语、韩语等所用的表意文字以外）。用字节表示黑白图像中的灰度也是很合适的，因为人眼大约能区分256种不同程度的灰度。当一个字节不足以表示信息时（如刚才说的表意语言：汉语、日语、韩语等），用两个字节，即 2_{16} 或65 536也可以很好地表示。

半个字节，即4位二进制数，有时被称为半位元组，它比起字节而言，用的并不频繁。由于字节在计算机内部经常出现，所以尽可能简单明了地表示它会带来很大方便。例如，

一个8位二进制数10110110的确很直观但是不简明。当然也可以用十进制数来表示字节，但这要求从二进制换算成十进制，这样做不仅不简

单，反而是件很令人讨厌的事。第8章曾描述了一种很直观的方法。每一位二进制数字写到对应的方盒子中，并在其下方标上2的乘方数。把每一列相乘后再相加即可得到对应的十进制数。下面是10110110的转换：

1 0 1 1 0 1 1 0

$\times 128$ $\times 64$ $\times 32$ $\times 16$ $\times 8$ $\times 4$ $\times 2$ $\times 1$

$$128 + 0 + 32 + 16 + 0 + 4 + 2 + 0 = 182$$

把十进制数转换为二进制数就更麻烦了。你可以用十进制数去除以递减顺序排列的 2 的 幂，每除一次，商是一个二进制位，而余数则继续去除以下一个最大的 2 的 幂。下面是十进制 数182转换成对应二进制数的过程：

182	54	54	22	6	6	2	0
-----	----	----	----	---	---	---	---

$\div 128$	$\div 64$	$\div 32$	$\div 16$	$\div 8$	$\div 4$	$\div 2$	$\div 1$
------------	-----------	-----------	-----------	----------	----------	----------	----------

1	0	1	1	0	1	1	0
---	---	---	---	---	---	---	---

第8章有关于这个方法的更多描述。总之，在十进制数和二进制数之间进行转换通常不是

件十分简单的事。

从第8章中我们还学习了八进制数字系统，八进制数字系统只使用数字 0、1、2、3、4、5、6、7。在八进制数和二进制数之间进行转换却是小菜一碟，你只要记住每个八进制数字对应 3 位二进制数字即可，如下表所示：

二进制	八进制
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

如果已有一个二进制数字（如 10110110），则从最右边的数字开始，每 3 位二进制数字组

成一组，即可转换为对应的八进制数：

$$\begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} & \underbrace{\hspace{1cm}} & & & & & \\ 2 & 6 & 6 & & & & & \end{array}$$

可见，字节 10110110 可以表示为八进制的 266。这显然已简单了很多，八进制确实是表示字节的一个好方法，但其中仍然有一个问题。

字节的二进制表示范围是从 00000000~11111111，对应的八进制表示范围是从 000~377。上例清楚地表示出 3 位二进制数对应于最右边和中间的八进制数，而 2 位二进制数对应于最左

边的八进制数，这就表明一个 16 位二进制数的八进制表示和把它分成两个字节后的八进制表示有所不同：

$$\begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline & & & & & & & & & & & & & & & \\ 1 & 3 & 1 & 7 & 0 & 5 \end{array}$$
$$\begin{array}{ccc} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ \hline 2 & 6 & 3 \end{array} \quad \begin{array}{ccc} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 3 & 0 & 5 \end{array}$$

为了使多字节值能和单字节值的表示一致，需要的系统应该能使每个字节平分，这意味着应该把每个字节分成 4 组，每组 2 位(以 4 为基数)；或 2 组，每组 4 位(以 16 为基数)。

让我们看看以 16 为基数的情况，这是我们还未接触过的新的记数系统。它被称为“十六进制 (hexadecimal)”，这个词本身就让人迷惑，因为大部分以 hexa- 为前缀的词都是指与 6 有关的事物，而这里 hexadecimal 却是指 16。虽然微软公司在技术出版物的格式方面明确地声明不要将十六进制缩写为 hex，但绝大多数人还是使用这种缩写。

在十进制中，我们这样计数：

0 1 2 3 4 5 6 7 8 9 10 11 12 ……

在八进制中，不需要 8和9:

0 1 2 3 4 5 6 7 10 11 12

同样，以 4 为基数的系统不需要 4、5、6 或 7:

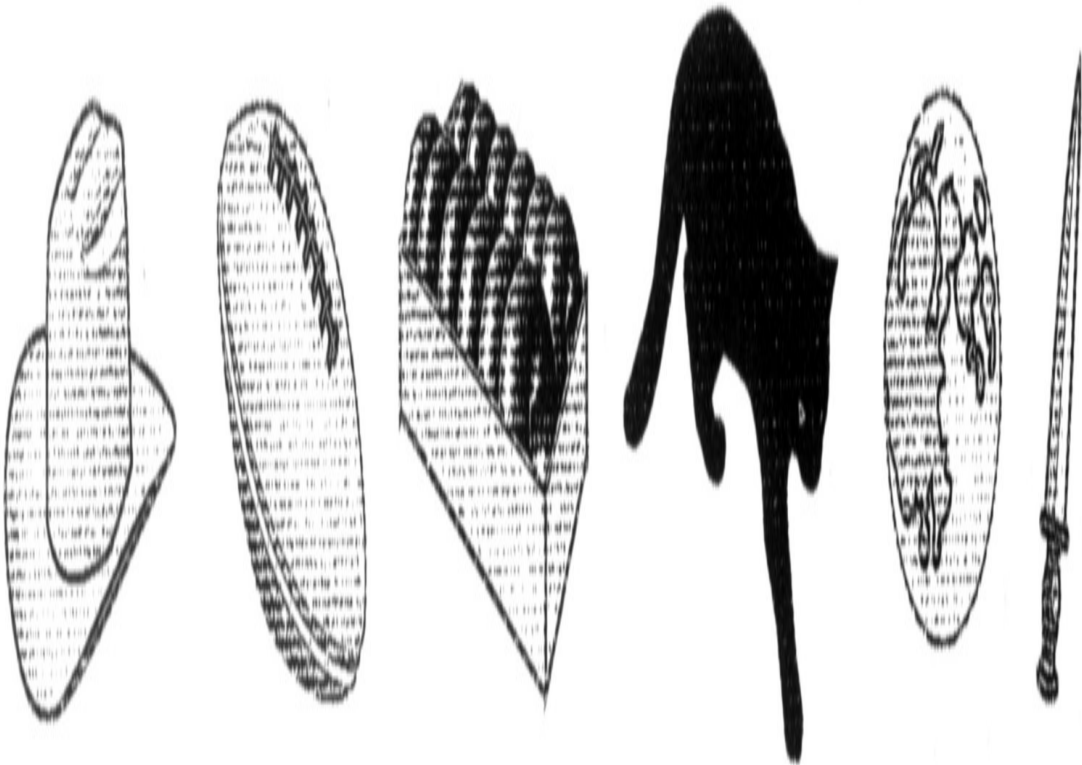
0 1 2 3 10 11 12

而二进制只需要 0和1:

0 1 10 11 100

但是十六进制有些不同，它需要的符号比十进制还要多，如下所示：







0 1 2 3 4 5 6 7 8 9 ? ? ? ? ? ? 10 11 12



上图中 10出现的地方是表示十进制中的数 16。此外，还需要 6个符号来表示十六进制数，但这些符号是什么呢？它们来自哪里呢？最形象的方法是引入 6个新符号，例如：

它们不同于大部分数字用的符号，这些符号的优点是便于记忆，而且从某种意义上代表了它们应该表示的数字。你看，重 10 加仑的牛仔帽、一个橄榄球（一个橄榄球队有 11 人）、一打椰子、一只黑猫（和不吉利的 13 相联系）、一轮满月（一般出现在新月后的第 14 天晚上）以及让人们联想到 Julius Caesar 在三月的第 15 天被暗杀时所用的匕首。

每个字节都可以用两个十六进制数表示，换句话说，一个十六进制数代表 4 位二进制数，即半个字节。下表描述了在二进制数、十六进制数和十进制数之间的转换：

Binary	Hexadecimal	Decimal	Binary	Hexadecimal	Decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010		10
0011	3	3	1011		11
0100	4	4	1100		12
0101	5	5	1101		13
0110	6	6	1110		14
0111	7	7	1111		15

二进制数

十六进制数

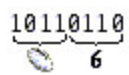
十进制数

二进制数

十六进制数

十进制数

10110110



下图表示如何用十六进制表示字节 10110110:

即使要表示多字节数也很容易:



一个字节总是由一对十六进制数来表示。

不过，我们绝不会真的用橄榄球或匕首来表示十六进制数，事实上，我们用 6 个拉丁字母 来表示那 6 个十六进制数，如下所示：

0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 ……

下表表示出在二进制数、十六进制数和十进制数之间的转换：

二进制数	十六进制数	十进制数
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

字节**10110110**可以由十六进制数 **B6**来表示，而不用再画上一个橄榄球。同前面的章一样，

用下标来表示记数系统的基数，如： 10110110

表示二进制、2312

FOUR

表示四进制、 266

EIGHT

表

示八进制、 182

TWO

表示十进制、 B6

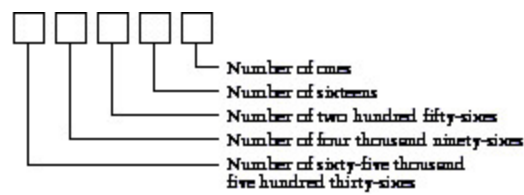
则表示十六进制。不过，这真是挺麻烦的。还好，有

更简单、更通用的方法来表示十六进制: **B6**

HEX

还可以进一步简化为 **B6h**。

在十六进制数中， 每一位的位置都对应于 16的幂：



1 的个数

16 的个数

256 的个数

4096 的个数

65 536 的个数

十六进制数 9A48Ch是:

$$9A48Ch = 9 \times 10000h +$$

$$A \times 1000h + 4 \times 100h +$$

$$8 \times 10h + C \times 1h$$

用16的乘方表示为:

用对应的十进制数代入为:

$$\begin{aligned}
 9A48Ch &= 9 \times 16_4 + \\
 A \times 16_3 &+ 4 \times 16_2 + \\
 8 \times 16_1 &+ C \times 16_0
 \end{aligned}$$

$$\begin{aligned}
 9A48Ch &= 9 \times 65536 + \\
 A \times 4096 &+ 4 \times 256 + \\
 8 \times 16 &+ C \times 1
 \end{aligned}$$

注意，在写一个数时，不用下标来表示数的基数也不会引起混淆。9就是9，不管它是十进制数还是十六进制数；而A则显然是个十六进制数，相当于十进制中的10。

把所有数字转换成十进制数需要下列运算：

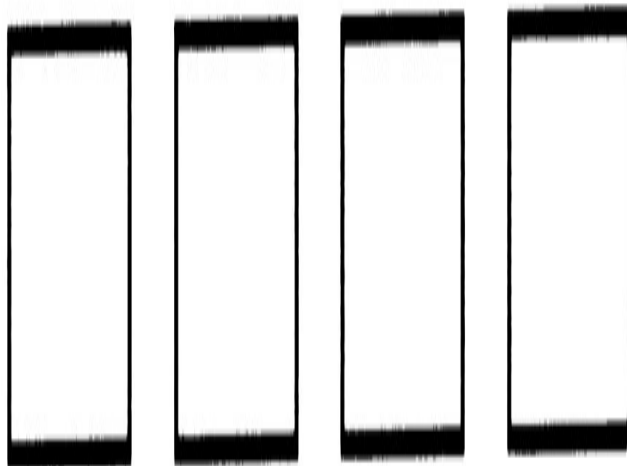
$$9A48Ch = 9 \times 65536 +$$

$$10 \times 4096 +$$

$$4 \times 256 +$$

$$8 \times 16 +$$

$$12 \times 1$$



x4096

x256

x16

x1

$$\square + \square + \square + \square = \square$$

答案是**631 948**。以上是一个十六进制数如何转换成十进制数的过程。下面是把任何一个 4 位十六进制数转换成十进制数的模板：

例如，这儿有一个79ACh的转换过程。记住，十六进制中的A和C对应于十进制中的10和12：

把十进制数转换为十六进制数需要做除法。如果数字比 255小，则可以用 1个字节来表示，对应于两个十六进制数。为了求出这两个数，用原数去除以 16得到商和余数。举例说明， 182 除以16得11，余6，所以十六进制表示为 B6h。

如果想要转换的十进制数比 65 536小，则十六进制表示会有 4位或更少。下面是把这样一个十进制数转换为十六进制数的一个模板：

-4096	-256	-16	-1

31,148	2476	172	12
-4096	-256	-16	-1
7	9	10	12

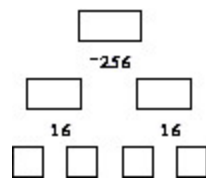
先把整个十进制数放到左上角的盒子里，作为第一个被除数，用它除以 4096（第一个除数），商放到被除数下面的盒子里，余数放到被除数右边的盒子里。余数成为新的被除数，用它除以 256。以下是 31 148如何转换成十六进制数的过程：

当然，十进制数的 10和12用十六进制表示就是 A和C，故结果是 79ACh.

这个方法的问题是如果你想用一个计算器来做除法运算，它不会显示出余数是多少。如果你用31 148 除以4096，计算器给出的结果只能是 7.6044921875。为了计算余数你得用 4096×7

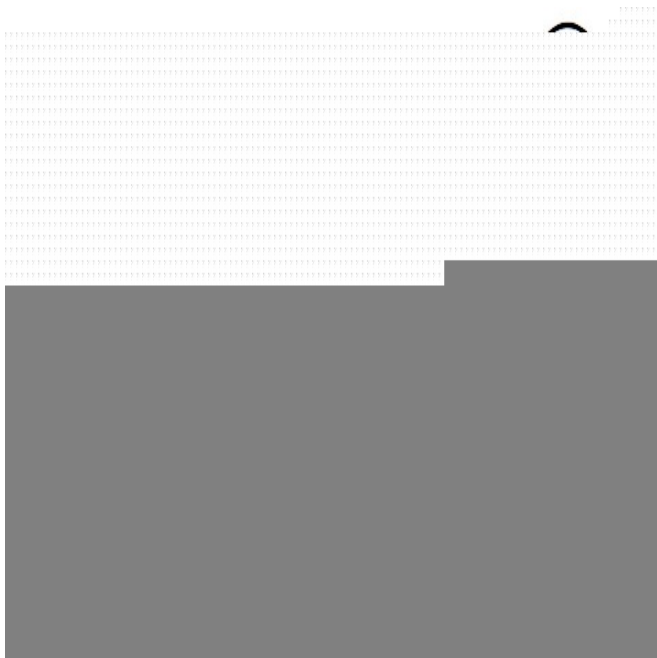
(得到 28 672)，再从 31 148中减去它；或者用 4096乘以 0.6044921875，即商的小数部分。（不过，有些计算器具有十进制数和十六进制数之间的转换功能。）

把小于 65 535的十进制数转换为十六进制数的另一个方法是先把原数除以 256而分为两个 字节，对于每个字节，再除以 16。下面是模板：



从最上面开始，每做完一次除法，商就进入除数左下方的盒子里，而余数进入右边的盒 里里。例如， 51 966的转换过程是：

得到的十六进制数字是12、10、15和14，即CAFE，它看起来倒更像一个单词而非一个数字！下面是和十六进制相关的加法表：



可以用这张表和通常的进位规则来对十六进制数做加法：

$$\begin{array}{r}
 4A3378E2 \\
 + 877AB982 \\
 \hline
 D1AE3264
 \end{array}$$

曾在第 13 章中讲过可以用 2 的补数来表示负数。如果在二进制中处理 8 位带符号数，则所有的负数都是以 1 开头的。在十六进制中，两位带符号数若是以 8、9、A、B、C、D、E 或 F 开头则是负数。例如，99h 可能表示十进制的 153（如果处理的是 1 个字节的无符号数）或十进制的 -103（如果处理的是有符号数）。

字节99h也有可能就是十进制的 99，关于这一点会在第 23章中解释，但是下一步必须先讲 讲存储器。

第 16 章 存储器组织

每天早上，当我们从睡梦中醒来时，记忆会填充大脑的空白。我们会记起我们在哪里，做过什么，计划做什么。我们可能一下子就能想起来，也可能几分钟都想不起来，不过，总的来说，我们通常能够重新组织自己的生活，保持高度的连续性，开始新的一天。

当然，人类的记忆是无序的。当回忆高中的几何课时，你可能会想到是谁坐在你前面；或者那一天当老师要解释什么是 QED(*quod erat demonstrandum*，证完/证毕)的时候，刚好进行消防演习。

人类的记忆也非安全无比。其实，书写的发明就是为了弥补人类记忆的不足。前一天晚上你可能因为突然冒出的一个关于剧本的好主意而在凌晨三点醒来，抓起床边特地准备的笔和纸记下来以便不会忘掉，第二天早上你就可以看到这个好主意并开始着手写剧本。当然你也可以不用这样。

先写后读，先保存后取回，先存储后访问，存储器的作用就是在这两类事件间保证信息的完好无损。无论什么时候存储信息，都要用到不同类型的存储器。纸是保存文本信息的最佳媒体，磁带则能很好地保存音乐和电影。

电报继电器—当集成为逻辑门然后再集成为触发器—也一样可以保存信息。正如我们所知道的，一个触发器可保存 1 位信息。保存 1 位信息当然并不代表保存全部信息，但这是一个开端。一旦我们知道了如何存储 1 位信息，就可以容易地存储 2 位、3 位或更多位信息。

第 14 章曾讲过电平触发的 D 型触发器，它由一个反向器、两个与门和两个或非门构成：



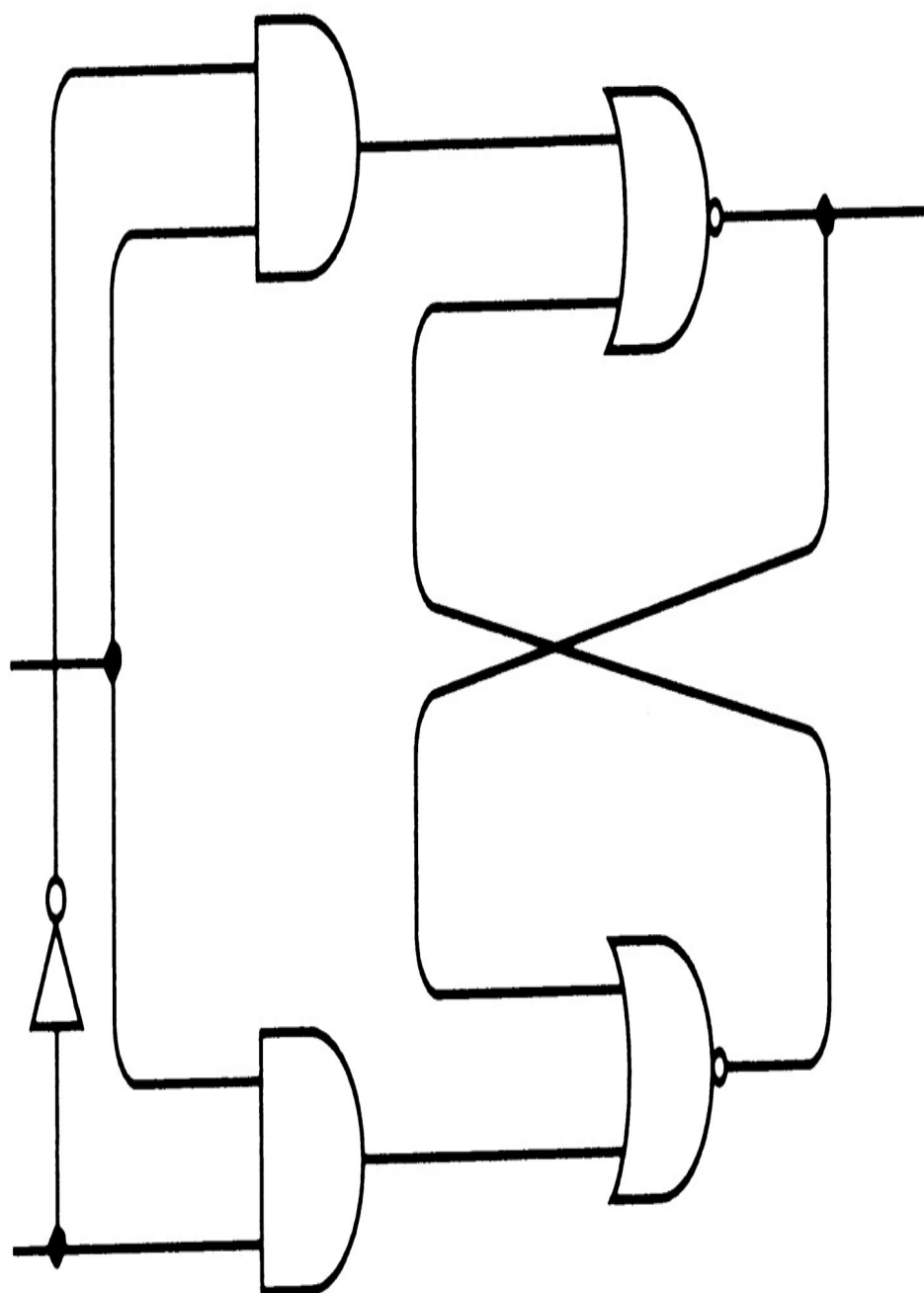
数据端

当时钟输入为1时，Q端输出与数据端输入是相同的。但当时钟输入变为0时，Q端输出将 维持原来的数据端输入，再改变数据端输入不会影响Q端输出，直到时钟输入再次变为1为止。触发器的逻辑表格如下：

Inputs		Outputs
D	Clk	Q
0	1	0
1	1	1
X	0	Q

输入 输出

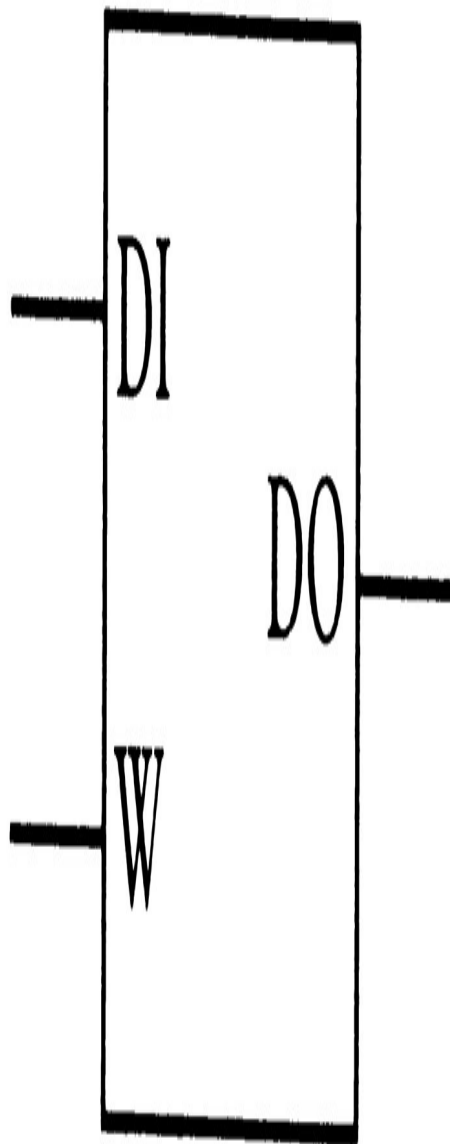
在第14章中，这种触发器的功能体现在两个不同的电路中。而在本章，它仅以一种方式来使用——即用于保存1位信息。正因为如此，给输入端和输出端重新命名，以便与该目的更为一致。



数据输出

写入

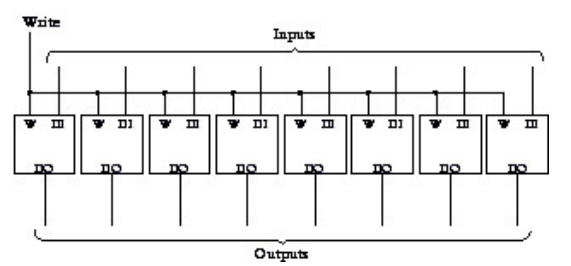
数据输入



这是同一个触发器，但现在 Q 输出端命名为数据输出（Data Out），时钟输入端（在第 14 章是作为保持位）命名为 写入（Write）。就像可以在纸上记录信息一样，写入信号使得数据输入（Data In）信号写入或存储到电路中。通常，若写入信号 (W) 为 0，则数据输入 (DI) 信号对输出无影响。而当我们想在触发器中存储数据输入信号时，写入信号应先置 1 后置 0。就像 在第 14 章提到

的，这种类型的电路也叫锁存器，因为它锁定数据。下面画出了一个 1 位锁存器， 但没有画出其所包含的单个部件：

把多个1位锁存器连成多位锁存器是相当容易的， 只需连接好写入信号：

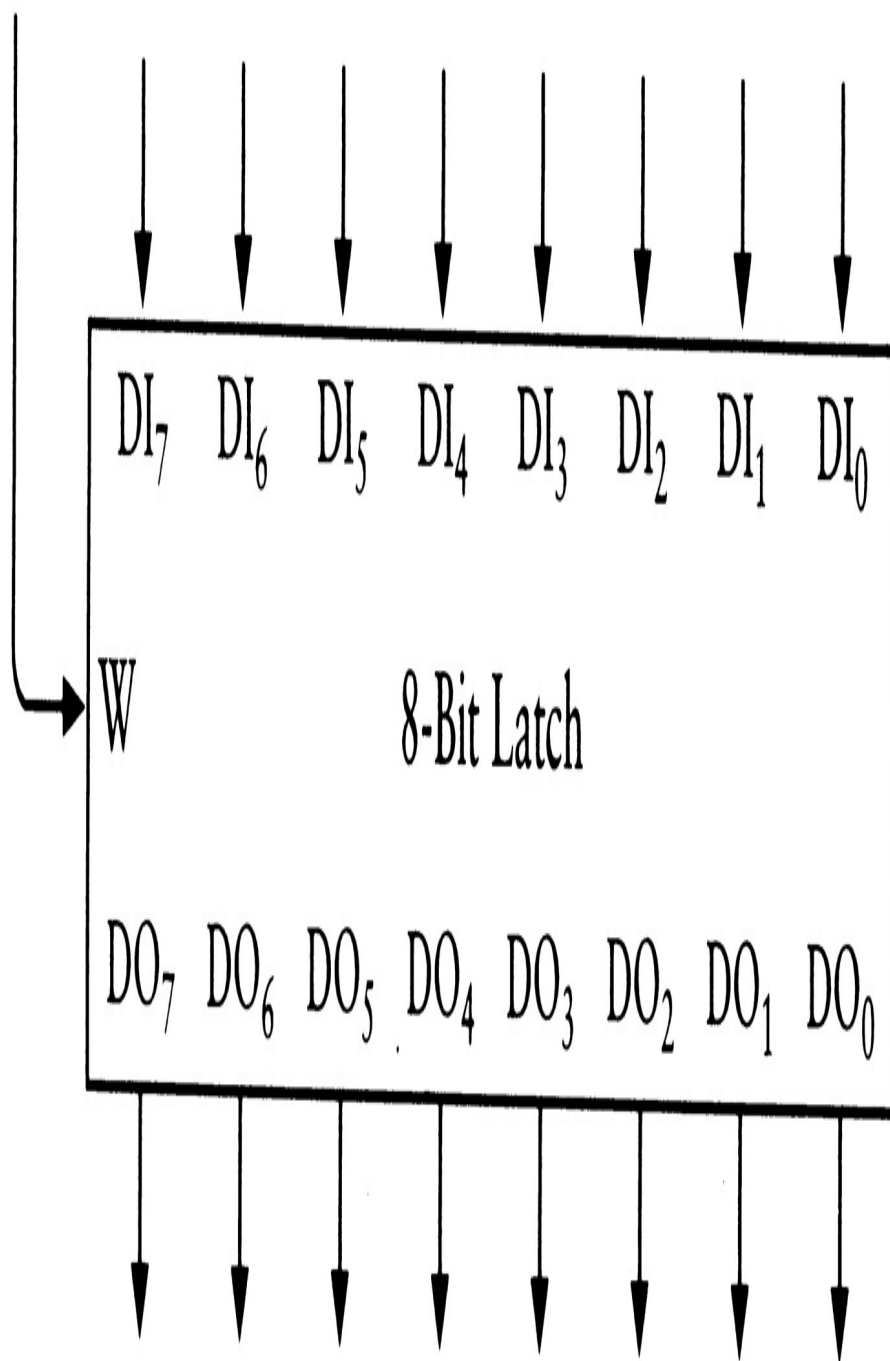


写入

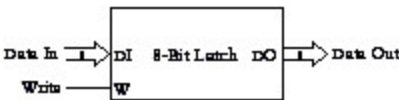
输入

输出

该8位锁存器具有 8个输入端和 8个输出端。另外，这个锁存器有一个写入输入端，通常为 0。要在这个锁存器中存储一个 8位二进制数，应将写入信号先置 1后置0。也可以把这个锁存器画成一个整体，就像这样：



为了与1位锁存器一致，也可以画成这样：



数据输入

8 位锁存器

数据输出

写入

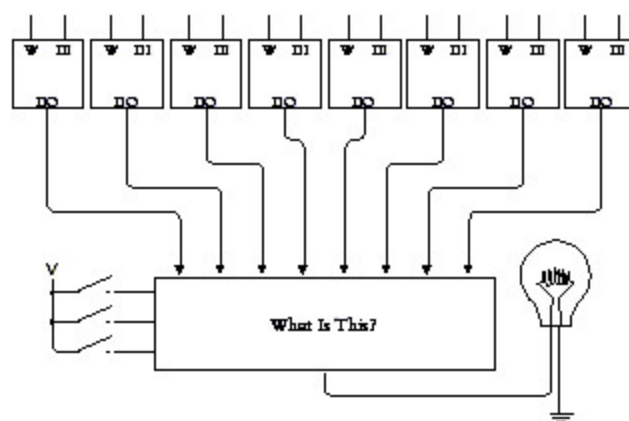
另外一种集成 8 个 1 位锁存器的方法不像上述这么直接。假设只想用一个数据输入信号端 和一个数据输出信号端，且又希望它具备在一天或一分钟内存储 8 次数据输入信号的能力。同时，也希望能够通过检测这个数据输出信号端就可以检查这 8 个数据。

换句话说，我们只想存储 8 个单独的 1 位数，而不想在 8 位锁存器中存储 1 个 8 位数。为什么会有这种想法呢？可能是因为我们仅有一个灯泡的缘故吧！

我们知道这需要 8 个 1 位锁存器。先不要考虑这些数据是怎样存储在这些锁存器中的，先 让我们把注意力放在如何用一个灯泡来检查 8 个锁存器的数据输出信号上。当然，我们通常用 手工把灯泡从一个锁存器移到另一个锁存器来测试各个锁存器的输出，不过，我们更倾向于 用更自动化的方法来实现。实际上，我们打算用开关来选择想要检查的锁存器。

那么，需要多少个开关呢？若是 8 个锁存器，则需要 3 个开关。3 个开关可表示 8 个不同的 值：000、001、010、011、100、101、110 和 111。

目前已有 8 个 1 位锁存器、3 个开关、1 个灯泡，此外还有“其他东西”用在开关和灯泡之间：

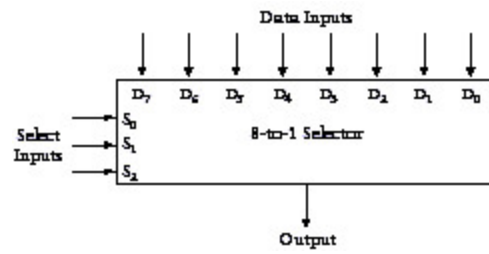


这是什么？

这个“其他东西”就是标识为“这是什么？”的神秘盒子，它上面有 8 个输入端，左侧有 3 个输入端。通过闭合和断开这三个开关，就可以从 8 个输入中选择一个，使其经过底部至输

出端输出，该输出使灯泡发光。

“这是什么？”到底是什么呢？我们以前曾见过类似的东西，尽管没有这么多的输入端。它类似于第 14 章中第一个改进的加法机里用到的电路。那时我们需要某种东西用于选择一行开关还是选择一个锁存器的输出作为加法器的输入，我们把这种东西叫 2-1 选择器，这里需要 8-1 选择器：



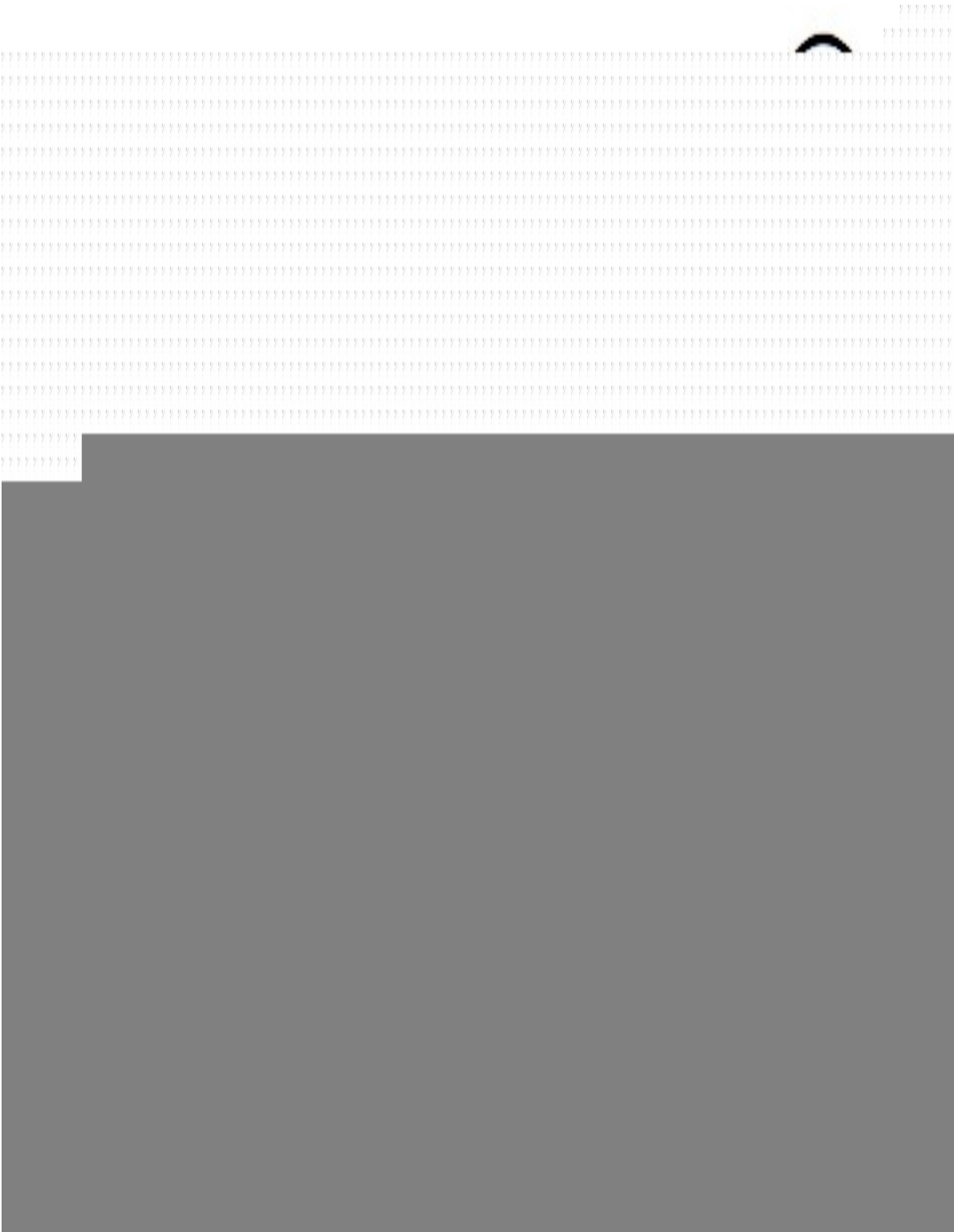
数据输入

选择输入

输出

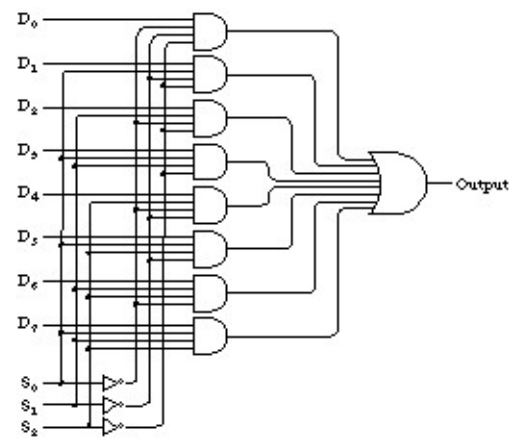
8-1选择器具有 8 个数据输入端（显示在上部）和 3 个选择输入端 (Select Input)（显示在左侧），选择输入端用于选择哪个输入数据在输出端输出。例如，若选择输入端为 000，则输出 D 的值；若选择端为 111，则输出 D 的值；若选择端为 101,则输出D 的值。其逻辑表如下：

0 7 5



输入 输出

8-1选择器由三个反向器、八个 4输入与门和一个 8输入或门构成，如下所示：



输出

这是一个相当复杂的电路，但只需一个例子就可以使你明白它是如何工作的。假设

$S=1, S=0, S=1$ ，从上面数第六个与门的输入包括 S 、 $\neg S$ 、 S ，它们全为 1。没有其他与门有同

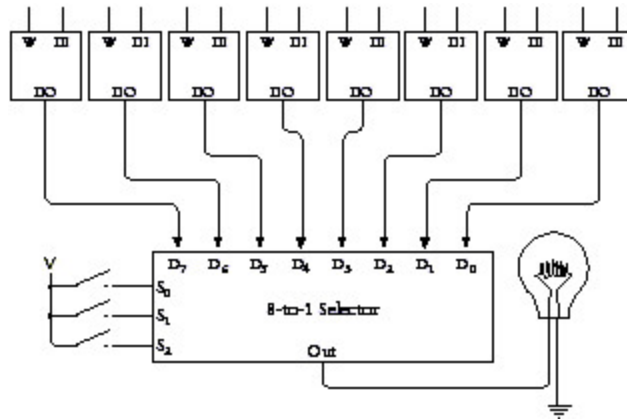
S

样的三个输入信号，因此，其他与门输出全部为 0。若 $D = 0$ ，则第六个与门输出为 0；若 $D =$

5 5

1，则其输出为 1。对最右边的或门来说也是如此。因此，若选择端为 101，则输出为 D 。

概括一下我们想干什么。我们想连接 8 个 1 位锁存器，它们能够通过一个数据输入信号端 分别写入，也能通过一个数据输出信号端 分别检查。已经证明可以用一个 8-1 选择器从 8 个锁 存器中选择一个数据输出信号，如下图所示：



8-1 选择器

输出

到现在已完成了任务的一半。我们已经实现了输出端的要求，现在再来看一下输入端。输入端包括数据输入信号及写入信号。在锁存器的输入端，可以把所有数据输入信号连

接在一起。但不能把 8 个写入信号也都连在一起，因为我们还想分别向每个锁存器中写入数据。此外，还要有一个单独的写入信号，它必须能连到其中任一个（并且只能是一个）锁存器上：



数据输入

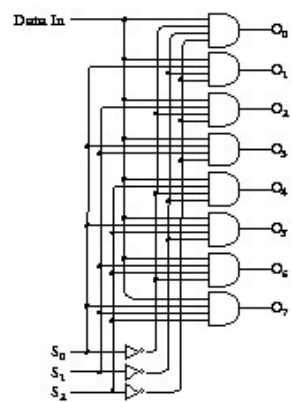
写入

这是什么？

为了完成这项工作，需要另外一个电路，这个电路看起来与 8-1 选择器有点相似，但实际上却正好相反。这就是 3-8 译码器。前面我们曾见过简单的数据译码器——第 11 章曾通过连接开关来选择理想的猫的颜色。

3-8 译码器有 8 个输出端。任何情况下，锁存器除了一个输出端外，其余的均为 0。这个例外是由 S_2 、 S_1 、 S_0 输入信号所选择的输出端。该输出端输出的也是数据输入端的输入：

0 1 2



数据输入

再说一遍，从上面数第六个与门的输入包括 S 、 $-$ 、 S ，没有另外的与门有同样的三个输

S

0 1 2

入。若选择输入端为 101，则其他与门输出都为 0。若数据输入为 0，则第六个与门输出为 0；

若数据输入为 1,则其输出为 1。其逻辑表格如下：

Inputs			Outputs							
s_2	s_1	s_0	O_7	O_6	O_5	O_4	O_4	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	Data
0	0	1	0	0	0	0	0	0	Data	0
0	1	0	0	0	0	0	0	Data	0	0
0	1	1	0	0	0	0	Data	0	0	0
1	0	0	0	0	0	Data	0	0	0	0
1	0	1	0	0	Data	0	0	0	0	0
1	1	0	0	Data	0	0	0	0	0	0
1	1	1	Data	0	0	0	0	0	0	0

输入

 输出

数据

 数据

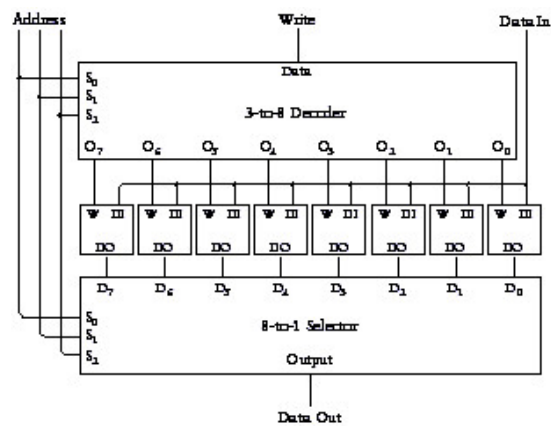
数据

 数据

数据 数据

数据 数据

下面是具有 8 个锁存器的完整电路：



地址 写入 数据输入

数据

3-8 译码器

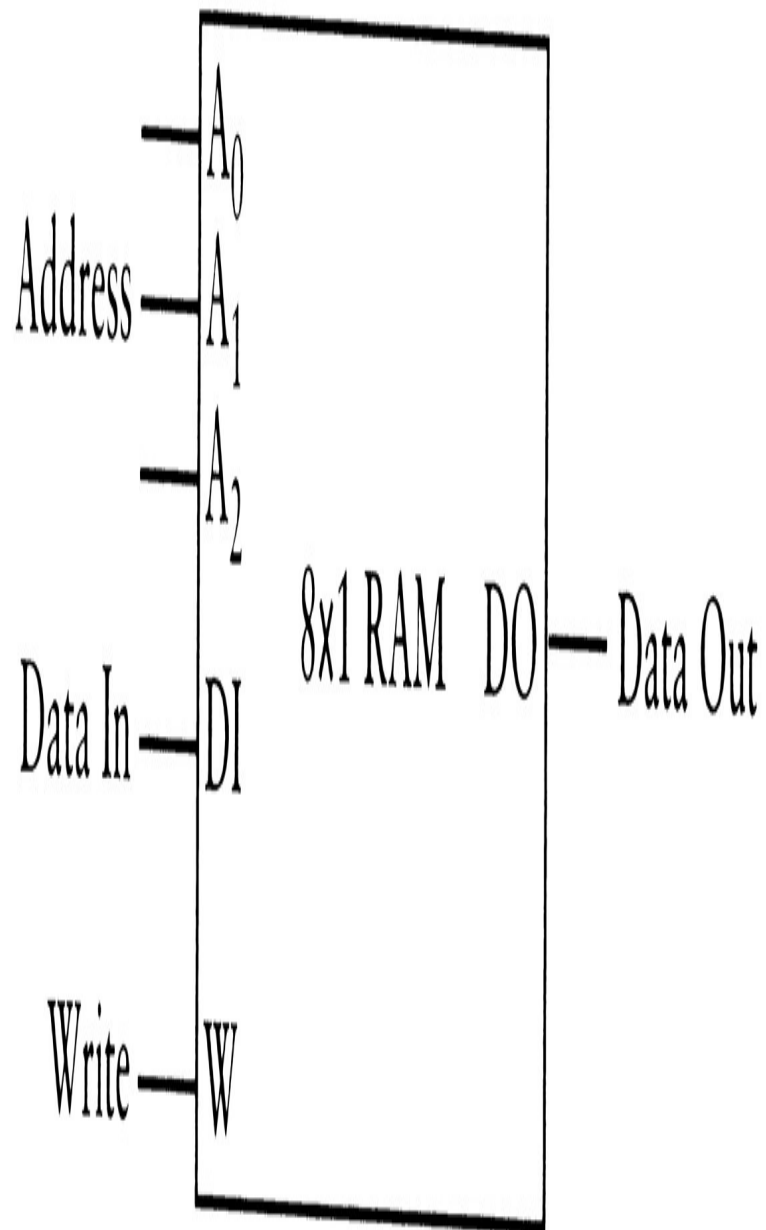
8-1 选择器 输出

数据输出

注意，译码器和选择器的三个选择信号相同，现在这三个信号都记作 地址（Address）。就像信箱号一样，3位地址决定了选择 8个锁存器中的哪一个。在输入端，地址输入决定写入 信号触发哪一个锁存器来存储输入的数据。在输出端（图的下部），地址输入控制 8-1选择器 选择8个锁存器中的一个进行输出。

这种锁存器的配置有时也称为读／写存储器，但通常叫作随机访问存储器或 RAM。RAM

可存储8个单独的 1位数据，如下图所示：



数据输入

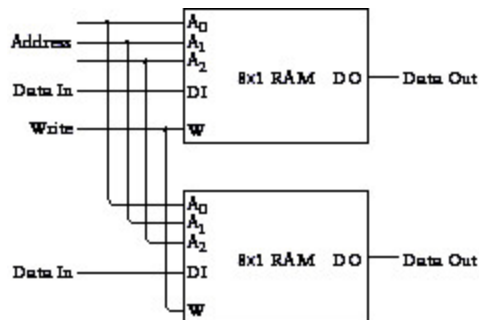
数据输出

写入

称它为存储器是因为它能保存信息，称为读／写存储器是因为可以在每个锁存器中保存新的数据（也就是写数据），同时还可以查看每个锁存器中所保存的数据（也就是读数据）。称它为随机访问存储器是因为通过简单地改变地址输入就可以从 8 个锁存器中的任意一个读出或写入数据。相比之下，其他类型的存储器必须顺序读出——也就是，在可以读出存储在地 址 101 的数据之前，必须读出存储在地址 100 的数据。

RAM 配置通常称作 RAM 阵列，上述这种特定配置的 RAM 阵列以简写形式 8×1 的方式组织起来。阵列中可以存放 8 个数，每个仅占 1 位，RAM 阵列中能存储的位数等于这两个值的乘积。RAM 阵列可通过各种方法来组合。例如，可以把两个 8×1 RAM 阵列连接起来，使它们

按照相同的方法来寻址：



地址

数据输入

数据输出

数据输入

数据输出

这两个 8×1 RAM 阵列的地址和写入输入端连接在一起，所以其结果为一个 8×2 RAM 阵列：

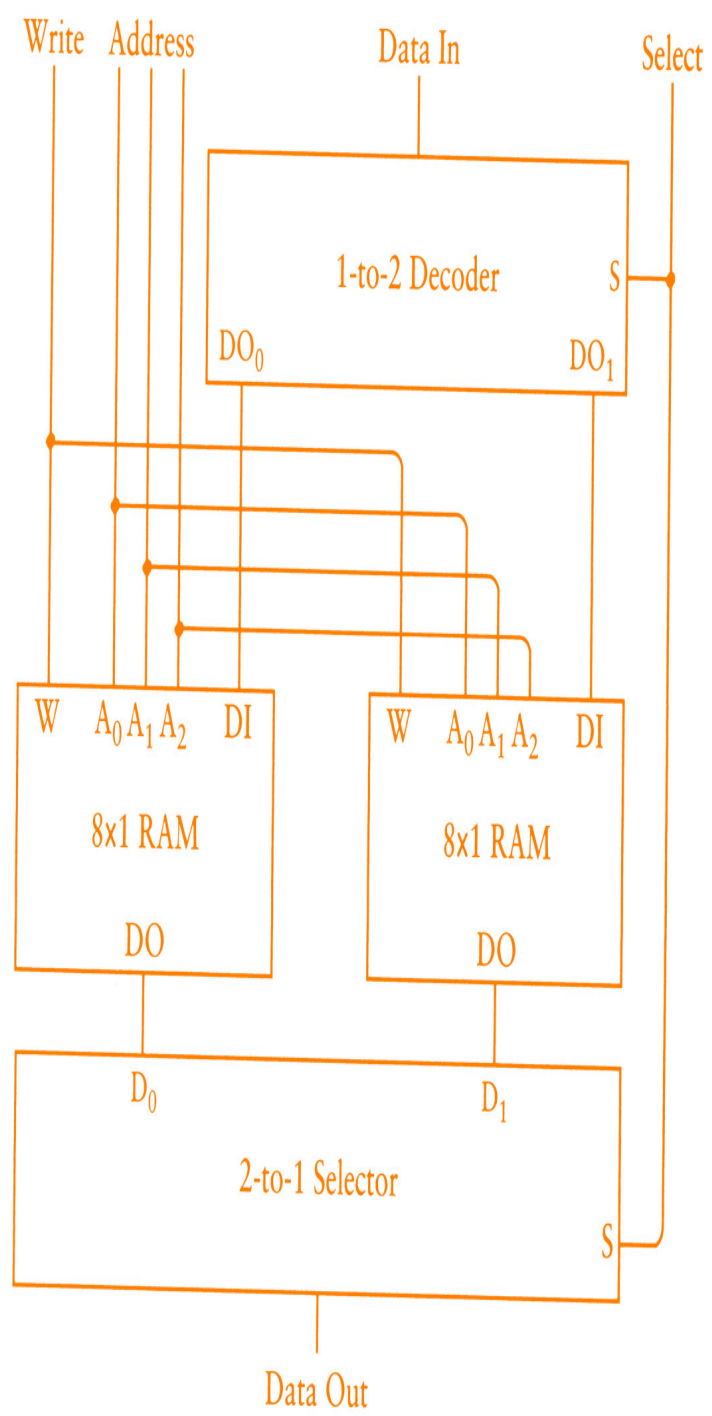
地址

数据输入

数据输出

这个RAM阵列可存储 8 个数，但每个数占 2 位。

两个 8×1 RAM 阵列也可以按照与单个锁存器连接相同的方式—通过一个 2-1 选择器和一个 1-2 译码器—来组合，如下图所示：



写入 地址 数据输入 选择

1-2 译码器

2-1 选择器

数据输出

连接到译码器和选择器的选择（**Select**）输入实质上选择两个 **8×1 RAM** 阵列中的一个， 在这里它也就是第 4 根地址线。所以，该图实际上是一个

16×1 RAM 阵列:

地址

数据输入

写入

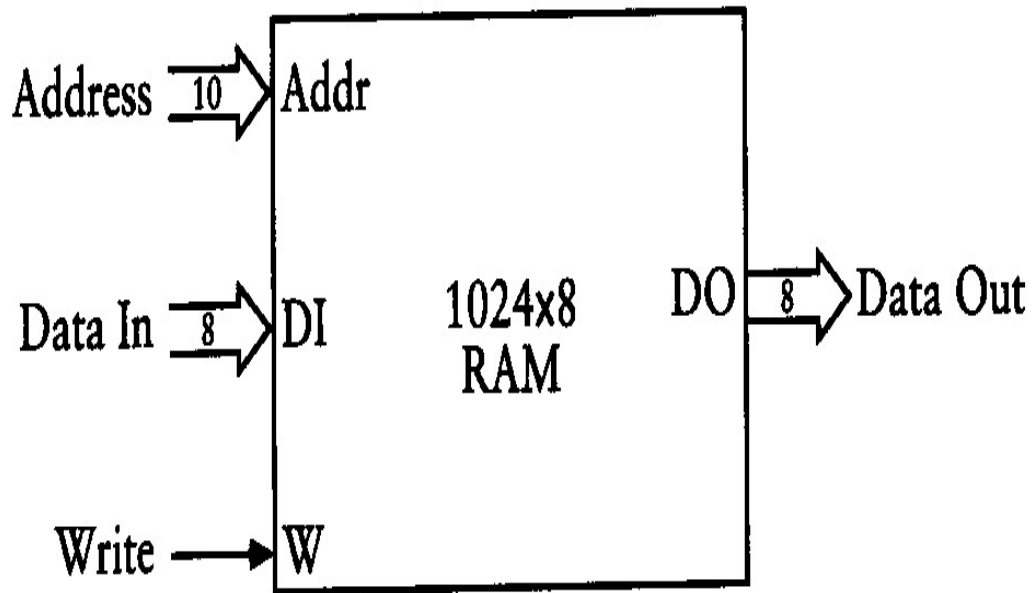
数据输出

此RAM阵列可存储 16个数，每个数占 1位。RAM阵列的存储容量与地址输入端数目有直接关系。无地址输入端时（即 1位锁存器和 8

位锁存器这种情况），只能存1个数；有一个地址输入端时，可存 2个数；有两个地址输入端时，可存 4个数；有三个地址输入端时，可存 8个数；有四个地址输入端时，可存 16个数。其关系可归纳成如下等式：

$$\text{RAM 阵列的存储容量} = 2^{\text{地址输入端数目}}$$

上面已讲了如何构造小的 RAM 阵列，那么再规划大的 RAM 阵列应该并不困难。例如：



地址

数据输入

数据输出

这个RAM阵列可存储 8192位信息，按 1024个数、每个 8位来组织。因为 $1024 = 2_{10}$ ，所以它有10条地址线。此外，它有 8个数据输入端和 8个数据输出端。

换句话说，这个 RAM 阵列可存储 1024个字节。就像一个邮局有 1024个邮箱，每个信箱中 有一个不同的 1字节数。

1024个字节即 1K 字节（kilobyte），1K字节在此会引起许多混淆。公制里前缀 k（来自于 希腊文 khilioi, 意思为1千）经常用到，如 1kg=1000g, 1km=1000m。但这里所说的 1K字节=1024 字节——而非1000字节。

原因在于公制是基于 10的幂，而二进制是基于 2的幂，这两种进制永远不会有相同的值。10的整数次幂为 10、100、1000、10000、100000等，而 2的整数次幂为 2、4、8、16、32、64 等，没有 10的整数次幂与 2的整数次幂相等的情况。

但有时它们非常接近。的确，1000非常接近 1024，可以用“约等于 (\approx)”符号进行数学化表示：

$$2_{10} \approx 10_3$$

这个关系式并非不可思议，它只不过表明 2的某次幂等于 10的某次幂而已。这种特例允许 人们方便地把 1024字节称作 1K字节。

K字节简写为 K或KB。所以，上面展示的 RAM阵列存储 1024字节或1K(1KB)。

不能把 1KB的RAM阵列说成是存储 1000字节，它大于 1000，即 1024，为了让人知道你在 说什么，你可以把它说成“1K”或“1K字节”。

1K字节的存储器具有 8 个数据输入端， 8 个数据输出端和 10 个地址输入端。由于是通过 10 条地址线来访问字节，所以 RAM 阵列可存储 2_{10} 个字节。无论何时再加上一条地址线，其存储容量将翻倍。下面每一行都都表示存储容量的翻番：

$$1\text{KB} = 1024\text{B} = 2_{10} \text{B} \approx 10_3 \text{B}$$

$$2\text{KB} = 2048\text{B} = 2_{11} \text{B}$$

$$4\text{KB} = 4096\text{B} = 2_{12} \text{B}$$

$$8\text{KB} = 8192\text{B} = 2_{13} \text{B}$$

$$16\text{KB} = 16\,384\text{B} = 2_{14} \text{B}$$

$$32\text{KB} = 32\,768\text{B} = 2_{15} \text{B}$$

$$64\text{KB} = 65\,536\text{B} = 2_{16} \text{B}$$

$$128\text{KB} = 131\,072\text{B} = 2_{17} \text{B}$$

$$256\text{KB} = 262\,144\text{B} = 2_{18} \text{B}$$

$$512\text{KB} = 524\,288\text{B} = 2_{19} \text{B}$$

$$1024\text{KB} = 1\,048\,576\text{B} = 2_{20} \text{B} \approx 10_6 \text{B}$$

可以看出左侧的数字也是 2 的整数次幂。

按照同样的逻辑，我们能把 1 0 2 4 字节称作 1 K B，当然也可以把 1 0 2 4 K B 称作 1 M 字节

(megabyte, 希腊文 megas 意思为大)，M 字节缩写为 MB。以下仍是存储容量翻番的式子：

$$1\text{MB} = 1\,048\,576\text{B} = 2_{20} \text{B} \approx 10_6 \text{B}$$

$$2\text{MB} = 2\,097\,152\text{B} = 2_{21} \text{B}$$

$$4\text{MB} = 4\,194\,304\text{B} = 2_{22}\text{ B}$$

$$8\text{MB} = 8\,388\,608\text{B} = 2^{23} \text{ B}$$

$$6\text{MB} = 16\,777\,216\text{B} = 2^{24} \text{ B}$$

$$32\text{MB} = 33\,554\,432\text{B} = 2^{25} \text{ B}$$

$$64\text{MB} = 67\,108\,864\text{B} = 2^{26} \text{ B}$$

$$128\text{MB} = 134\,217\,728\text{B} = 2^{27} \text{ B}$$

$$256\text{MB} = 268\,435\,456\text{B} = 2^{28} \text{ B}$$

$$512\text{MB} = 536\,870\,912\text{B} = 2^{29} \text{ B}$$

$$1024\text{MB} = 1\,073\,741\,824\text{B} = 2^{30} \text{ B} \approx 10^9 \text{ B}$$

希腊文gigas意思为巨大，所以把 1024MB称作1G字节(gigabyte)，缩写为GB。

同样，1T字节(terabyte，希腊文 teras意思为庞然大物)等于 2^{40} 字节（约 10^{12} ）或1 099 511 627 776B,terabyte缩写为TB。

1KB约为1000B，1MB约为1 000 000B,1GB约为1 000 000 000B,1TB约为1 000 000 000 000B。

再大的数就很少用了，如 1PB（petabyte）= 2^{50} B或1 125 899 906 842 624 字节，约等于

10^{15} 。1EB(exabyte)= 2^{60} B或1 152 921 504 606 846 976字节，约等于 10^{18} 。

下面提供一些基本常识。在此书编写的时候（1999年），家用电脑一般都配有 32MB或 64MB或128MB的随机访问存储器（为不至于混淆，这里不谈任何关于硬盘驱动器的事情，而只谈论RAM），即33 554 432B或67 108 864B或134 217 728B。

当然，人们总拣方便的讲。有 65 536 字节内存的人会说“我有 64K”；有33 554 432字节

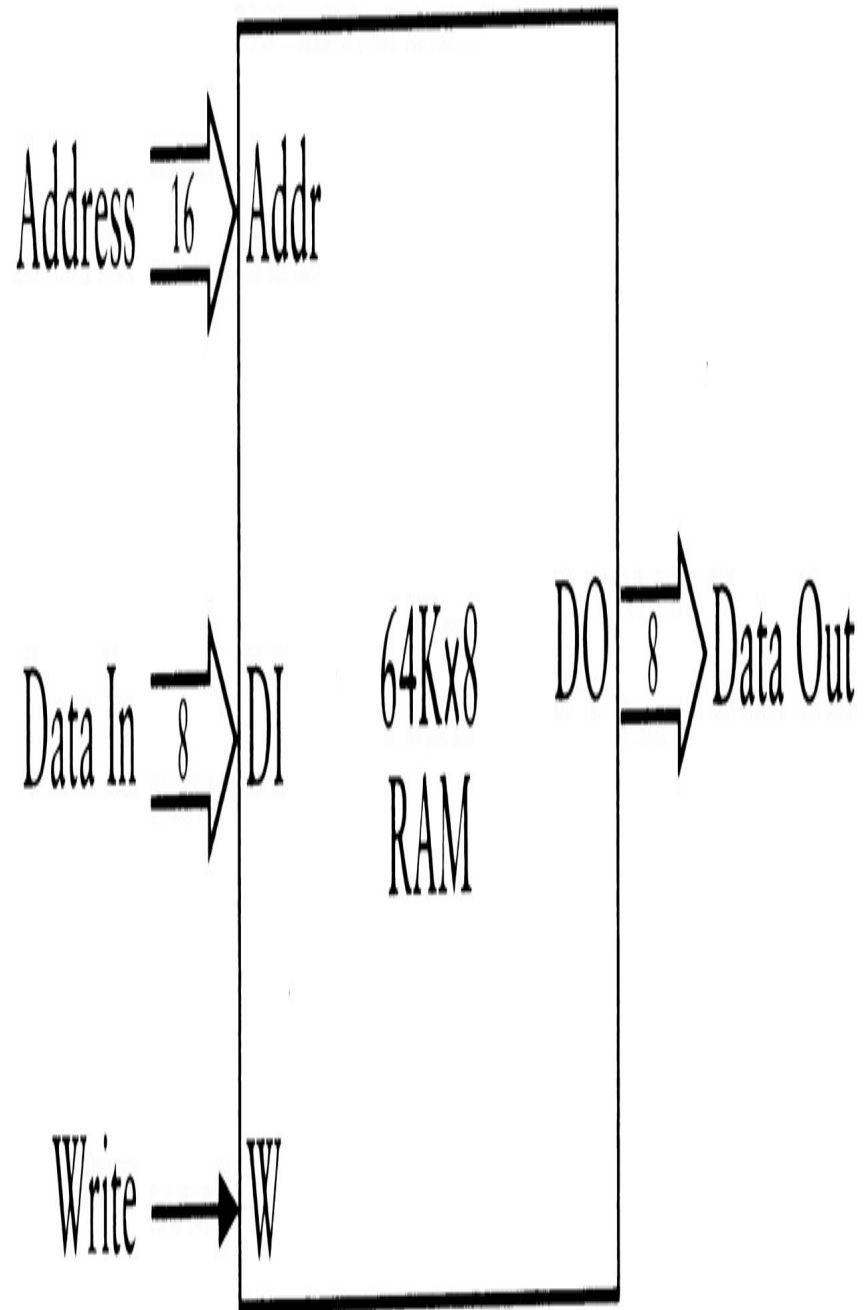
的人会说“我有 32M”。虽说不多，但有 1 073 741 824字节的人也会说“我有 1G”。有时人们可能会提到 K位或M位（注意是位而不是字节），不过这很少见。人们谈到存储

器时，几乎总是指字节数而非位数。（当然，把字节转换成位，乘 8 即可。）在线路传送数据时，

通常会有这样的短语每秒千位(kbps)或每秒兆位(mbps)出现。例如， 56K的调制解调器指的是

56Kbps,而非每秒千字节。

至此我们已经明白如何构造所需的 RAM阵列，但不要离题太远。现在让我们看一下已经集成了65 536字节的存储器：



地址

数据输入

数据输出

写入

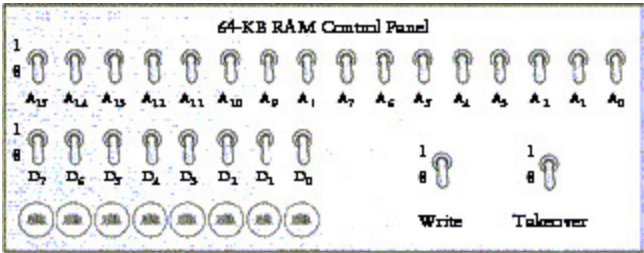
为什么是64KB而非32KB或128KB？因为65 536是一个整数，刚好为 2_{16} B ，也即该RAM阵列有16位地址。换句话说，该地址正好是2个字节。用十六进制来表示其地址范围是 0000h～

FFFFh.

64KB的内存在 1980年的PC机上是比较普遍的配置，尽管它不是用电报继电器制成的。但是，你真的能用继电器来实现吗？肯定不能。因为按照我们的设计方案需要为每位存储器提供9个继电器，那么 64K×8的RAM阵列需要将近 500万个继电器。

利用控制面板来操作所有的存储器—写入数据到存储器或验证写入的数据—将更加先进。这种控制面板用 16个开关来表示地址，8个开关来表示需要输入存储器的 8位数据，8个灯

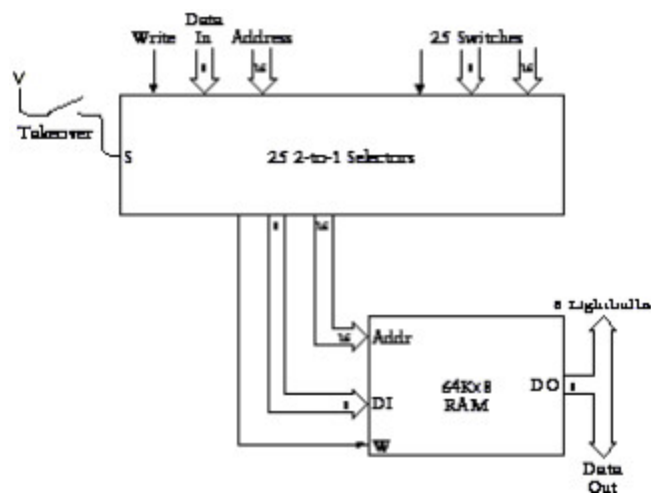
泡来显示 8 位数据，再用一个开关来表示写入信号，如下图所示：



控制面板

所有的开关均显示在 0 位置。另外，还有一个标识为接管 (takeover) 的开关，使用这个开关 的目的是使其他电路可以使用与 控制面板相连的同一个存储器。当接管开关置 0 时（如图所 示）， 控制面板上的其余开关将不起任何作用；而当此开关置 1 时，控制面板将对存储器进行专门控 制。

这些都可以用若干 2-1 选择器来实现。实际上，需要 25 个—16 个 接地址信号、 8 个接数据 输入开关、另外 1 个接写入开关。其电 路如下：



数据

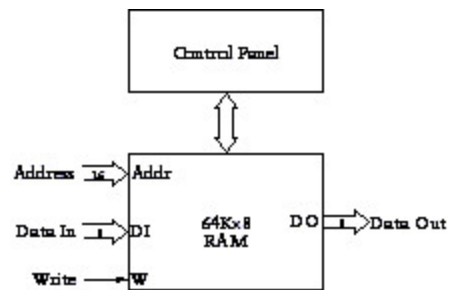
接管

25 个 2-1 选择器

8 个灯泡

当接管开关断开时（如上图），64K×8 RAM阵列的地址、数据输入和写入信号来自于外部信号，显示在2-1选择器的左上角；当接管开关闭合时，RAM阵列的地址、数据输入和写入信号来自于控制面板开关传来的信号。无论哪种情况，RAM阵列的数据输出信号传到8个灯泡上或其他可能的地方。

下图是带有控制面板的64K×8 RAM阵列：



控制面板

地址

数据输入

数据输出

写入

当接管开关闭合时，可用 16 个地址开关来选择 65 536 个地址中的任何一个，而灯泡将显

示当前地址中所存的 8 位数据。可用 8 个数据开关来定义一个新数，并通过写入开关把它写入 存储器中。

通过 64K×8 RAM 阵列和控制面板可以与需要处理的 65 536 个 8 位数据中的任何一个保持联系。但我们也留了一些机会让别的东西——也许是其他一些电路——使用存在存储器中的数据，或者把数据写入存储器。

还有一个必须注意的有关存储器的问题，它非常重要。在第 11 章介绍逻辑门的概念时，并未画出构成逻辑门的单个继电器的构造。特别地，没有标出每个继电器连接的电源。任何时候当继电器触发时，电流流过电磁线圈并在适当的位置吸下金属簧片。

如果一个装满 65 536 字节的 64K×8 RAM 阵列被关掉电源，将会发生什么情况呢？所有的电磁铁将失去磁性，所有继电器的触点将回到未触发状态，RAM 中的内容也将永远丢失。

这就是随机访问存储器也称为易失性存储器的原因，它需要恒定的电源来保持其中的内容。

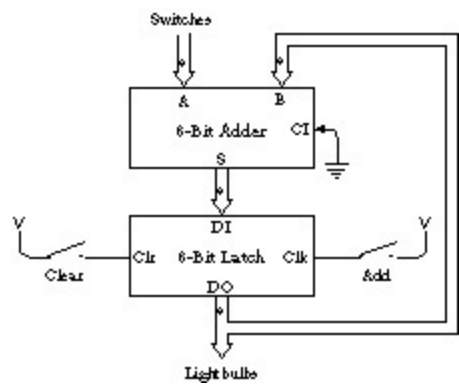
第 17 章 自动操作

人类是非常富于创造性而且是十分勤勉的，但是，人类在本质上也是十分懒惰的。非常明显，人类并不愿意去工作，这种对工作的反感导致人们用大量的时间来设计和制造可以把工作日缩短到几分钟的设备。幻想使人感到兴奋，甚至远比我们所看到新奇的事物更令人兴奋得多。

当然不会在这里介绍自动割草机的设计。本章将通过设计更精密的机器，使加减法运算更加自动化，这听起来也许有些不可思议。本章最后设计出的机器将具有广泛的用途，它实际上可以解决任何利用加减法的问题，这些问题的范围太大了。

当然，由于精密机器越来越复杂，因此有些部分可能会很粗糙。不过如果你略过了某些困难的细节，没有人会责备你。有时，你可能会不耐烦并且发誓再也不会因为一个数学问题而去寻求电或机械的帮助。不过请耐心坚持到底，因为本章最后将发明一个叫作计算机的机器。

我们曾在第 14 章见过一个加法器。它有一个 8 位锁存器，累加由 8 个开关输入的数的和：



开关

8 位加法器

8 位锁存器

清零 相加

前面曾讲过，8位锁存器用触发器来保存8位数据。使用这个设备时，必须首先按下清零开关使锁存器的存储内容清零，然后用开关来输入第一个数字。加法器简单地把这个数字与锁存器输出的零相加，因此其结果就是你刚输入的数字。按下相加开关可在锁存器中保存该数并且通过灯泡显示出来。现在从开关上输入第二个数，加法器把这个数与存储在锁存器中的数相加，再按下相加开关把总和存储在锁存器中并通过灯泡显示出来。通过这种方法，你可以加上一串数字并显示出运算总和。当然，其中存在的一个局限是8个灯泡不能显示总和超过255的数。

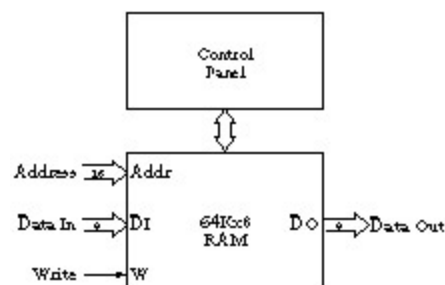
第14章介绍该电路的时候，只讲到一种锁存器，它是电平触发的。在电平触发的锁存器中，时钟输入端必须先置1然后回到0，才能使锁存器保存数据。当时钟信号等于1时，锁存器的数据输入可以改变，这种改变将会影响所保存的数据输出。第14章的后面又介绍了边沿触发的锁存器，这种锁存器在时钟输入从0变化到1的瞬间保存数据。由于边沿触发的锁存器易

于使用，所以假定本章用到的锁存器为边沿触发的锁存器。用于累加数字的锁存器叫作累加器，本章后面将会看到累加器并非仅仅进行简单的累加。

累加器通常是一个锁存器，保存第一个数字，然后该数字又加上或减去另一个数字。上面这个加法机存在的最大问题已经相当明显：如果想把 100 个二进制数加起来，你就得

坐在加法机前耐着性子输入每一个数字并累加起来。当你完成时，却发现有两个数字是错误的，你只好又重复全部的工作。

不过，也可能并非如此。上一章用了差不多 500 万个继电器来构造一个 64KB 的 RAM 阵列。另外，我们还连接了一个控制面板，用来闭合接管开关接通线路，并使用开关进行 RAM 阵列的写入和读出。



控制面板

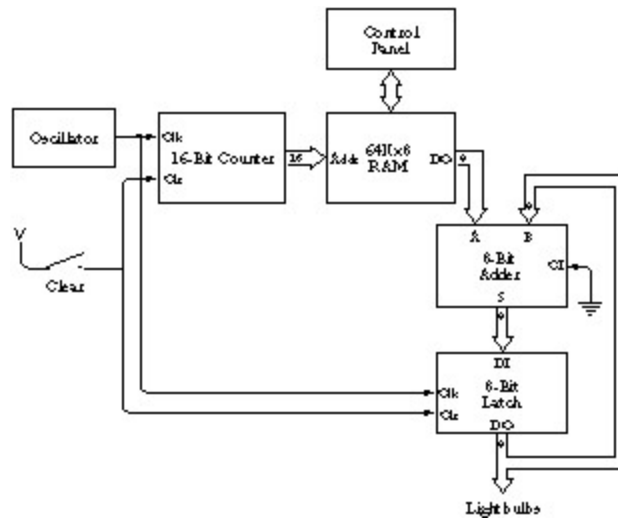
数据输入

数据输出

写入

如果你向 **RAM** 阵列中输入 100 个二进制数字，而不是直接输入到加法机中，那么进行数据修改会容易得多。

现在我们面临着一个挑战，即如何将 **RAM** 阵列连到累加器上。显而易见，**RAM** 的数据输出信号应该代替累加器的开关组。但是，用一个 16 位的计数器（正如在第 14 章构造的）就可以控制 **RAM** 阵列的地址信号。在下面这个电路中，连到 **RAM** 的数据输入信号和写入信号可以不要：



控制面板

振荡器

8位加法器

清零

8位锁存器

灯泡

当然这并非已经发明的最容易操作的计算装置。在使用之前，必须先闭合清零开关，以清除锁存器的内容并把16位计数器的输出置为0000h，接着闭合RAM控制面板上的接管开关。你可以从RAM地址的0000h处开始输入一组想要加的8位数，如果有100个数，则它们保存在从0000h~0063h的地址中（也可以把RAM阵列中没有用到的单元都设置为00h）。然后断开

RAM控制面板上的接管开关（这样控制面板不会再对 RAM阵列起控制作用了），并断开清零开关。这时，你就只需坐着看灯泡的亮灭变化了。

其工作情况为：当清零开关第一次断开时，RAM阵列的地址输入为 0000h，保存在 RAM

阵列当前地址的 8位数是加法器的输入。由于锁存器也清零，所以加法器的另 8位输入为 00h。振荡器提供时钟信号——一个在 0和 1之间迅速交替变化的信号。在清零开关断开后，当时

钟由 0变为 1时，将同时发生两个事件：锁存器保存来自加法器的结果；同时，16位计数器加 1，指向RAM阵列的下一个地址。在清零开关断开后，当时钟第一次由 0变为 1时，锁存器保存第一个数，同时，计数器增加到 0001h；当时钟第二次由 0变为 1时，锁存器保存第一个数与第二个数之和，同时计数器增加到 0002h；依此类推。

当然，这里先做了一些假设，首要一点，振荡器需慢到允许电路的其余部分可以工作。

对于每次时钟振荡，在加法器输出端显示有效和之前，许多继电器必须触发其他继电器。这种电路有一个问题，即没有办法让它停止。到一定时候，灯泡会停止闪动，因为 RAM

阵列中的其余数都为 00h。这时，你可以看到二进制和。但当计数器最终到达 FFFFh时，它又会翻到 0000h（就像汽车里程表），这时自动加法器又会开始把这些数加到已经计算过的和中。

这种加法机还有一个问题：它只能用于加法，并且只能加 8位数。不仅在 RAM阵列中的每

个数不能超过 255，而且其总和也不能超过 255。这种加法器也没有办法进行减法运算。虽然可以用 2 的补码表示负数，但是在这种情况下，加法器只能处理 -128~127 之间的数字。让它处理更大数字（例如，16 位数）的一种显而易见的方法就是使 RAM 阵列、加法器和锁存器的宽度加倍，同时再提供 8 个灯泡。不过你可能不太愿意做这种投资。

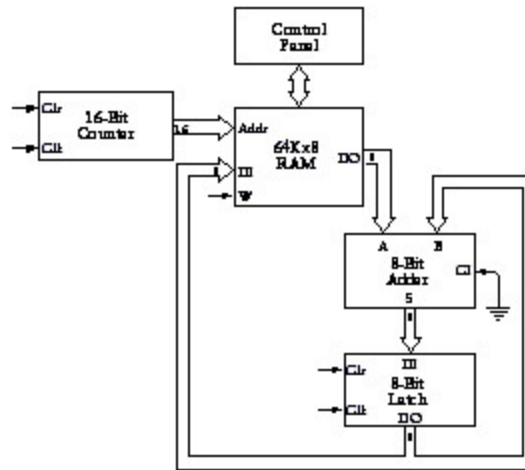
当然，要不是我们最终要去解决这些问题，这儿是不会提到这些问题的。不过我们首先

想谈的却是另外一个问题。如果不是要把 100 个数加成一个数，会怎么样？如果只想用自动加法器把 50 对数字加成 50 个不同的结果又会怎么样？也许你希望有一个万能的机器来累加多对数字、10 个数字或 100 个数字，并且希望所有的结果都可方便地使用。

前面提到的自动加法器在与锁存器相连接的一组灯泡上显示出其相加结果。对于把 50 对数字加成 50 个不同的和来说，这种方法并不好。你可能希望把结果存回 RAM 阵列中，然后，

在方便的时候用 RAM 控制面板来检查结果。控制面板上有专门为此目的而设计的灯泡。这意味着连接在锁存器上的灯泡可以去掉。不过，锁存器的输出端必须连接到 RAM 阵列

的数据输入端上，以便于和可以写入到 RAM 中：



控制面板

16 位计数器

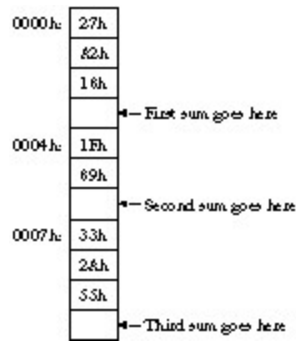
8 位加法器

8 位锁存器

上图中省略了自动加法器的其余部分，特别是振荡器和清零开关，因为不再需要显著标出计数器和锁存器的清零和时钟输入来源。此外，既然我们已经充分利用了 **RAM** 的数据输入端，就需要有一种方法来控制 **RAM** 的写入信号。

我们不去考虑这个电路能否工作，而把重点放在需要解决的问题上。当前需要解决的问题是能配置一个自动加法器，它不会仅用来累加一串数字。我们希望能随心所欲地确定累加多少数字、在 **RAM** 中存储多少不同的结果以供日后检查。

例如，假设我们希望先把三个数字加在一起，然后把另两个数字加在一起，最后再把另外三个数加在一起。我们可能会将这些数字存储在从地址 **0000h** 开始的 **RAM** 阵列中，存储器的内容如下所示：



第一个和放在这里

第二个和放在这里

第三个和放在这里

这是本书第 16 章所说明的内容。方格里是存储单元中的内容，存储器的每一个字节在一个方格中。方格的地址在方格左面，并非每一个地址都要表示出来，存储器的地址是连续的，因而可以算出某个方格的地址。方格的右侧是关于这个存储单元的注释，它们表示出我们希望自动加法器在这些空格中存储三个结果。（虽

然这些方格是空的，但存储单元并非空的。存储单元中总有一些东西，即使只是随机数，但此时它不是有用的数。)

现在可以试一下十六进制算术运算并且把结果存到方格中，但这并不是此项试验的要点，我们想让自动加法器来做一些额外的工作。

不是让自动加法器只做一件事情——在最初的加法器中，只是把RAM地址中的内容加到称为累加器的8位锁存器中——实际上是让它做四件不同的事。要做加法，需先从存储器中传送一个字节到累加器中，这个操作叫作Load（装载）。第二项所要执行的操作是把存储器中的一个字节加(Add)到累加器中。第三项是从累加器中取出结果，保存(Store)到存储器中。最后，需要有一些方法使自动加法器停止(Halt)工作。

详细说来，让自动加法器所做的工作如下所示：

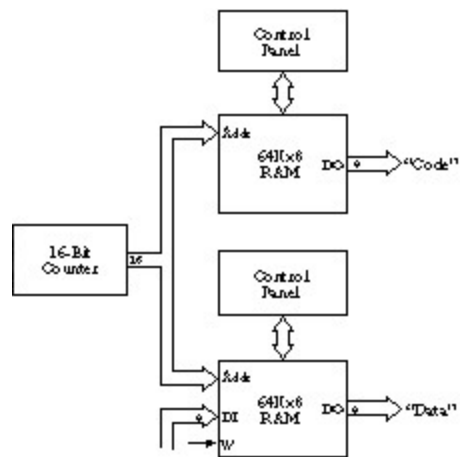
- 把地址0000h中的数装载到累加器中
- 把地址0001h中的数加到累加器中
- 把地址0002h中的数加到累加器中
- 把累加器中的数保存到地址 0003h中
- 把地址0004h中的数装载到累加器中

- 把地址0005h中的数加到累加器中
- 把累加器中的数保存到地址 0006h中
- 把地址0007h中的数装载到累加器中
- 把地址0008h中的数加到累加器中
- 把地址0009h中的数加到累加器中
- 把累加器中的数保存到地址 000Ah中
- 停止自动加法器的工作 注意，同最初的自动加法器一样，存储器的每个字节的地址是连续的，开始处为 0000h。

以前自动加法器只是简单地把存储器中相应地址的数加到累加器中。某些情况下，现在仍然 需要这样做，但有时我们也想直接把存储器中的数装载到累加器中或者把累加器中的数保存 到存储器中。在所有事情都完成以后，我们还想让自动加法器停下来以便检查 **RAM**阵列中的 内容。

怎样完成这些工作呢？只是简单地键入一组数到 **RAM**中并期望自动加法器来正确操作是 不可能的。对于 **RAM**中的每个数字，我们还需要一个数字代码来表示自动加法器所要做的工作：装载，加，保存或停止。

也许最容易（但肯定不是最便宜）的方法是把这些代码存储在一个完全独立的 **RAM**阵列 中。这第二个 **RAM** 阵列与最初的 **RAM** 阵列同时被访问，但它存放的不是要加的数，而是用 来表明自动加法器将要对最初的 **RAM**阵列的相应地址进行某种操作的代码。这两个 **RAM**阵列 可以分别标为数据（最初的 **RAM**阵列）和代码（新的 **RAM**阵列）：



控制面板

控制面板

已经确认新的自动加法器能够把“和”写入到最初的 RAM 阵列（标为数据），而要写入新的 RAM 阵列（标为代码）则只能通过控制面板来进行。

我们用 4 个代码来表示自动加法器希望能实现的 4 个操作。4 个代码可任意指定，下面为可

能的一组代码：

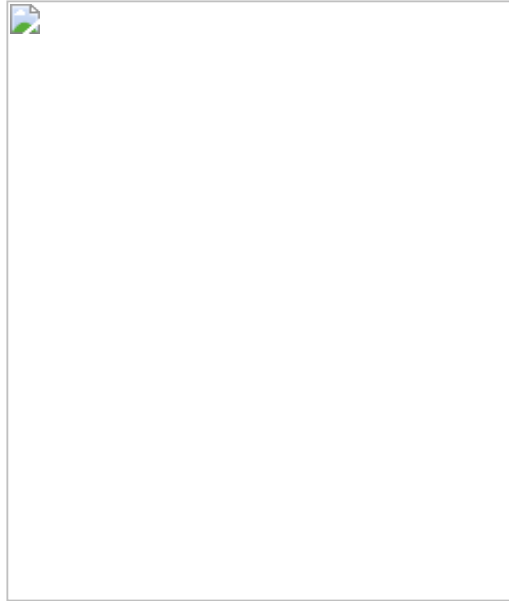
操作码	代码
Load (装载)	10h
Store (保存)	11h
Add (加)	20h
Halt (停止)	FFh

为了执行以上例子中提到的三组加法，需要用控制面板把下面这些数保存到代码**RAM**阵列中：

0000h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
0004h:	10h	Load
	20h	Add
	11h	Store
0007h:	10h	Load
	20h	Add
	20h	Add
	11h	Store
000Bh:	FFh	Halt

你可能想把这个 RAM 阵列中的内容与存放累加数据的 RAM 阵列的内容作一比较。你会发现代码 RAM 中的每个代码或者对应于数据 RAM 中一个要装入或加到累加器的数，或者表示一个要存回到存储器中的数。这样的数字代码通常称作指令码或操作码，它们“指示”电路执行某种“操作”。

前面谈到过，早期自动加法器的 8 位锁存器的输出需要作为数据 RAM 阵列的输入，这就是“保存”指令的功能。另外还需要一个改变：以前，8 位加法器的输出是作为锁存器的输入，但现在为了实现“装载”指令，数据 RAM 的输出有时候也要作为 8 位锁存器的输入，这种改变需要 2-1 数据选择器。改进的自动加法器如下图：



控制面板

代码

16位计数器

控制面板

数据

8 位加法器

2-1 选择器

8 位锁存器

上图中少了一些东西，但它显示了各种组件间的 8 位数据通路，一个 16 位计数器为 2 个 RAM 阵列提供地址。通常，数据 RAM 阵列输出到 8 位加法器上执行加法指令。8 位锁存器的输入可能是数据 RAM 阵列的输出也可能是加法器的输出，这需要 2-1 选择器来选择。通常，锁存器的输出又流回到加法器，但对“保存”指令而言，它又作为数据 RAM 阵列的输入。

上图中缺少的是控制这些组件的信号，统称为控制信号。它们包括 16 位计数器的时钟 (Clk) 和清零 (Clr) 输入，8 位锁存器的 Clk 和 Clr 输入，数据 RAM 阵列的写入 (W) 输入以及 2-1 选择器的选择 (S) 输入。其中有一些信号明显基于代码 RAM 阵列的输出，例如，若代码 RAM 阵列的输出表示装载指令，则 2-1 选择器的 S 输入必须为 0（选择数据 RAM 阵列的输出）。仅当操作码为保存指令时，数据 RAM 阵列的 W 输入才为 1。这些控制信号可以由逻辑门的各种组合来产生。

利用最小数量的附加硬件和新增的操作码，也能让这个电路从累加器中减去一个数。第 1

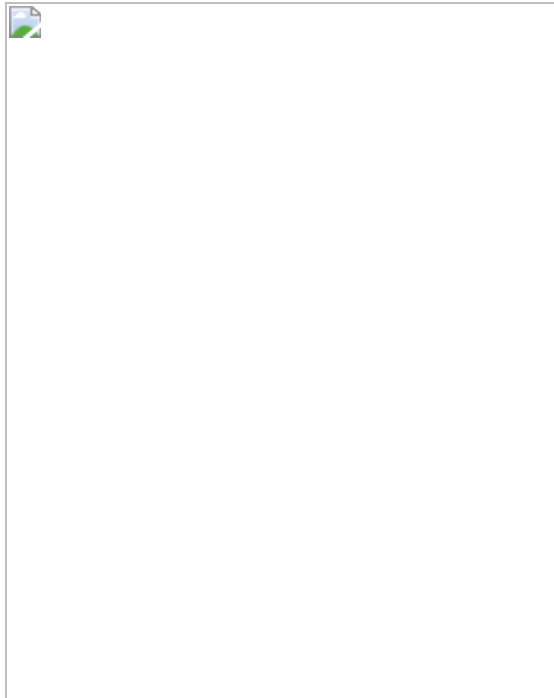
步是扩充操作码表：

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract(减)	21h
Halt	FFh

加法和减法只通过操作码的最低有效位来区分。若操作码为 21h，除了在数据 RAM 阵列 的输出数据输入到加法器之前取反并且加法器的进位输入置 1 外，电路所做的几乎与电路执行 加法指令所做的完全相同。在下面这个改进的有一个反相器的自动加法器里，C 信号可以完

0

成这两项任务：



控制面板

代码

16位计数

器 控制面板

数据

反相器

8位加法器

2-1 选择器

8位锁存器

现在假设把 56h和2Ah相加再减去 38h，可以按下图所显示的存储在两个 RAM阵列中的操 作码和数据进行计算：



“代码” “数据”

结果放在此处

装载操作完成后，累加器中的数为 56h。加法操作完成后，累加器中的数为 56h加上2Ah的和，即 80h。减法操作使数据 RAM阵列的下一个数（ 38h）按位取反，变为 C7h。加法器的进位输入置为 1时，取反的数 C7h与80h相加：

$$\begin{array}{r} \text{C7h} \\ + 80h \\ \hline + 1h \text{ 48h} \end{array}$$

其结果为 48h。（按十进制， 86加42减56等于72。）

还有一个未找到适当解决方法的问题是加法器及连到其上的所有部件的宽度只有 8位。以往唯一的解决方法就是连接两个 8位加法器（或其他的两个部件），形成16位的设备。

但也有更便宜的解决方法。假设要加两个 16位数，例如：

$$\begin{array}{r} 76\text{ABh} \\ + 232\text{Ch} \end{array}$$

这种16位加法同单独加最右边的字节（通常称作低字节）：

$$\text{ABh}$$

+ 2Ch

D7h

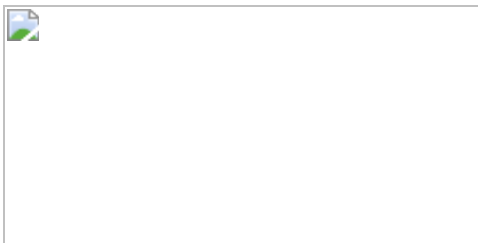
然后再加最左边的字节，即高字节

76h

+ 23h

99h

得到的结果一样，为 99D7h。因此，如果像这样把两个 16位数保存在存储器中：



“代码” “数据”

低字节结果

高字节结果

结果中的 D7h存在地址 0002h中， 99h存在地址 0005h中。当然，并非所有的数都这样计算，对上例中的数是这样计算。若两个 16 位数76ABh和

236Ch相加会怎么样呢？在这种情况下， 2个低字节数相加的结果将产生一个进位：

$$\begin{array}{r} \text{ABh} \\ + \text{6Ch} \\ \hline \end{array}$$

这个进位必须加到 2 个高字节数的和中：

最后的结果为 9A17h。

1h

+ 76h

+ 23h

9Ah

可以增强自动加法器的电路功能以正确进行 16位数的加法吗？当然可以。需要做的就是保存低字节数相加结果的进位，然后把该进位作为高字节数相加的进位输入。如何存储 1位 呢？当然是用 1位锁存器。这时，该锁存器称为进位锁存器。

为了使用进位锁存器，需要有另一个操作码，称作“进位加”（Add with Carry）。在进行 8位数加法运算时，使用的是常规“加法”指令。加法器的进位输入为 0，加法器的进位输出锁存在进位锁存器中（尽管根本不必用到）。

在进行 16位数加法运算时，仍然使用常规“加法”指令来进行低字节加法运算。加法器的进位输入为 0，其进位输出锁存到进位锁存器中。要进行高字节加法运算就要使用新的“进位加”指令。这时，两个数相加使用进位锁存器的输出作为加法器的进位输入。如果低字节加法有进位，则其进位可用在第二次运算中；如果无进位，则进位锁存器的输出为 0。

如果进行 16位数减法运算，还需要一个新指令，称为“借位减”（Subtract with Borrow）。通常，减法操作需要使减数取反且把加法器的进位输入置为 1。因为进位通常不是 1，所以往

往被忽略。在进行 16位数减法运算时，进位输出应保存在进位锁存器中。高字节相减时，进位锁存器的结果应作为加法器的进位输入。

加上新的“进位加”和“借位减”操作，共有 7个操作码：

操作码 代码

Load 10h

Store 11h

Add 20h

Subtract 21h

Add with Carry(进位加) 22h

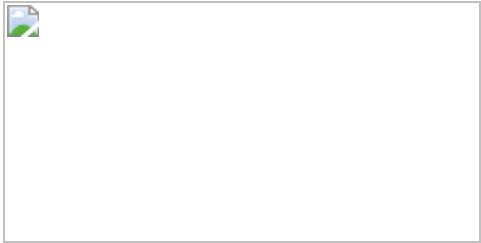
Subtract with Borrow(借位减) 23h

Halt FFh

在减法和借位减法运算中，需要把送往加法器的数取反。加法器的进位输出作为进位锁 存器的输入。无论何时执行加法、减法、进位加法和借位减法操作，进位锁存器都被同步。当进行减法操作，或进位锁存器的数据输出为 1 并且执行进位加法或者借位减法指令时， 8 位 加法器的进位输入被置为 1。

记住，只有上一次的加法或者进位加法指令产生进位输出时，进位加法操作才会使 8 位加

法器的进位输入为 1。任何时候进行多字节数加法运算时，不管是否必要，都应该用进位加法 指令计算。为正确编码前面列出的 16 位加法，可用如下所示方法：



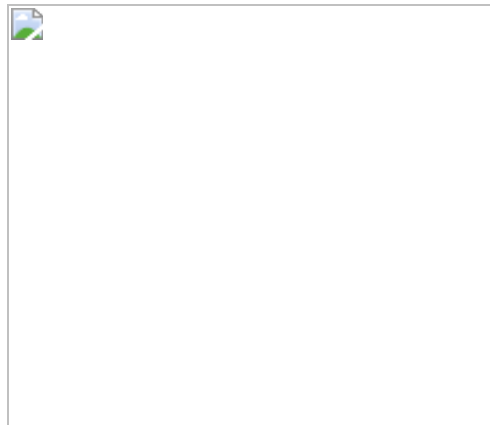
“代码” “数据”

低字节结果

高字节结果

不管是什么样的数，该方法都能正确工作。有了这两个新的操作码，极大地扩展了机器处理的范围，使其不再只局限于进行 8 位数加

法。重复使用进位加法指令，能进行 16 位数、 24 位数、 32 位数、 40 位数等更多位数的加法运算。假设要把 32 位数 7A892BCDh 与 65A872FFh 相加，则需要一个加法指令及三个进位加法指令：



“代码” 数据

低字节结果

次高字节结果

次高字节结果

最高字节结果

当然，把这些数存放到存储器中并非真的很好。这不仅要
用开关来表示二进制数，而且数在存储器中的地址也并不连续。例如，7A892BCDh从最低有效字节开始，每个字节分别存入存储器地址0000h、0003h、0006h及0009h中。为了得到最终结果，还必须检查地址0002h、0005h、0008h及000Bh中的数。

此外，当前设计的自动加法器不允许在随后的计算中重复利用计算结果。假设要把3个8位数加起来，然后再在和里减去一个8位数，并且存储结果。这需要一次装载操作、两次加法操作、一次减法和一次保存操作。但如果想从原先的和里减去另外一个数会怎么样呢？那个和是不能访问的，每次用到它时都要重新计算。

原因在于我们已经建造了一个自动加法器，其中的代码 **RAM**和数据 **RAM**阵列同时、顺序地从0000h开始寻址。代码 **RAM**中的每条指令对应于数据 **RAM**中相同地址的存储单元。一

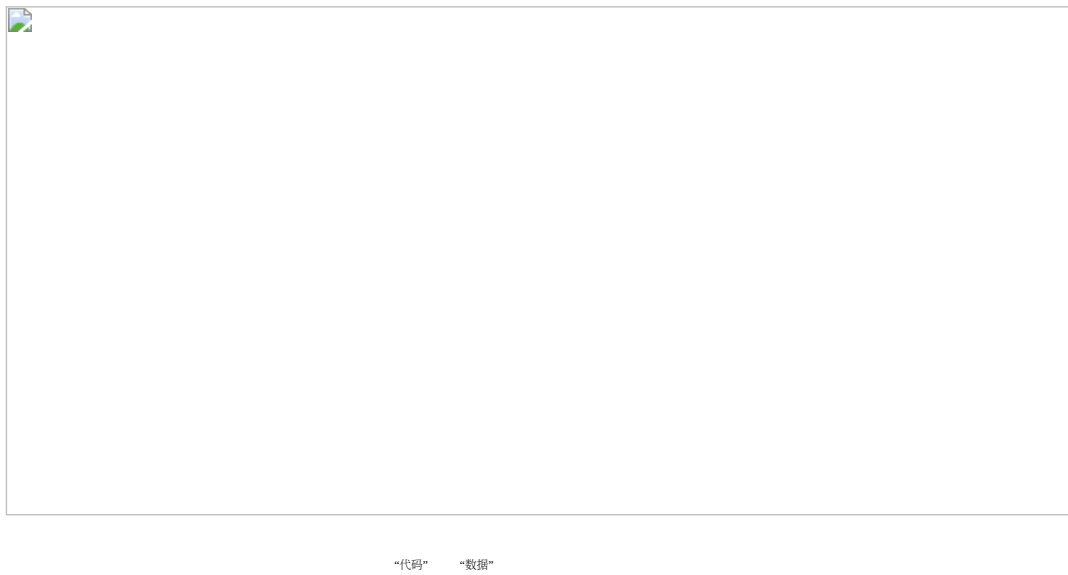
且“保存”指令使某个数据保存在数据 **RAM**中，这个数就不能再被装载到累加器中。为了解决这个问题，要对自动加法器做一个基本的及大的改变。虽说刚开始看上去会异

常复杂，但很快你就会看到一扇通向灵活性的大门打开了。让我们开始吧，目前我们已经有了7个操作码：

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry	22h
Subtract with Borrow	23h
Halt	FFh

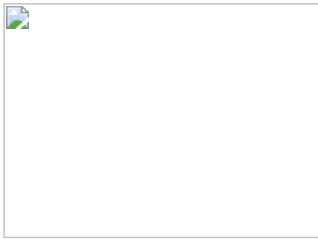
每个操作码在存储器中占 1个字节。除了“停止”代码外，现在希望每条指令在存储器中 占3个字节，其中第一个字节为代码本身，后两个字节存放一个 16位的存储器单元地址。对于 装载指令来说，其地址指明数据在数据 **RAM**阵列中的存储单元，该存储单元存放要装载到累 加器中的字节；对于加法、减法、进位加法和借位减法指令来说，地址指明要从累加器中加 上或者减去的字节的存储单元；对于保存指令来说，地址指明累加器中的内容将要保存的存 储单元。

例如，当前自动加法器所能做的最简单的工作就是加两个数。要完成这项工作，可以按 照下面的方法来设置代码 **RAM**阵列 和数据 **RAM**阵列：



结果

在改进的自动加法器中，每条指令（除了“停止”）需要3个字节：



“代码”

把地址 0000h 处的字节装入累加器

把地址 0001h 处的字节加到累加器

把累加器的内容存入地址 0002h 处

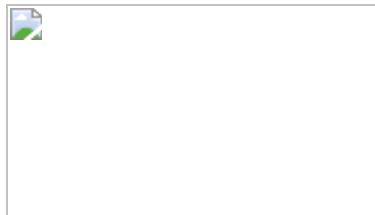
停止

每条指令（除了“停止”）后跟2个字节，用来表示在数据 RAM阵列中的 16位地址。这三个地址碰巧为 0000h、0001h和0002h，它们可以是任何其他地址。

前面说明了如何使用加法和进位加法指令来相加一对 16 位数——比如 76ABh 和 232Ch。必须把 2 个数的低字节保存在存储器单元 0000h 和 0001h 中，把 2 个高字节保存在 0003h 和 0004h 中，

其相加结果保存在 0002h和0005h中。这样，我们可以用更合理的方式来保存两个加数及其结果，这可能会保存在以前从未用

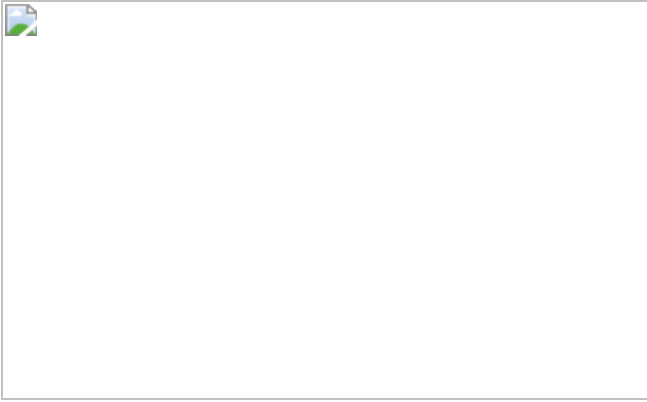
过的存储区域：



“数据”

结果的高字节放在这里 结果的低字节放在这里

这6个存储单元不必像图中这样连在一起，它们可分散在整个 64KB 数据 RAM阵列中的任 何地方。为了把这些地址中的数相加，必须在代码 RAM阵列中按如下所示设置指令：



“代码” “代码”

把地址 4001h 处的 字节装入累加器

把地址 4000h 处的字节装入 累加器

把地址 4003h 处的 字节加到累加器

把 4002h 处的字节同进位加到累加器

把累加器中的内容存入地址 4005h 处

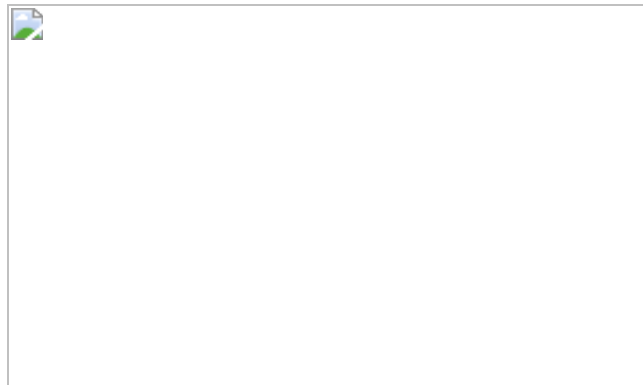
把累加器中的内容存入地址

4004h 处

停止

可以看到保存在地址 4001h和4003h中的两个低字节首先相加，并把结果保存在地址 4005h 中。两个高字节（在地址 4000h和4002h中）利用进位加法进行相加，其结果保存在地址 4004h 中。如果去掉“停止”指令并向代码 RAM中加入更多指令，随后的计算就可以简单地通过存储器地址来利用原先的数及它们的和。

实现这种设计的关键就是把代码 RAM阵列中的数据输出到3个8位锁存器中，每个锁存器保存3字节指令的一个字节。第一个锁存器保存指令代码，第二个锁存器保存地址的高字节，第三个锁存器保存地址的低字节。第二和第三个锁存器的输出组成了数据 RAM 阵列的16位地址：



8位 锁存器

控制面板

16位 计数器

8位

锁存 控制面板 器

8位 数据

锁存器

从存储器中取出指令的过程叫作取指令。在上述加法机中，每个指令长 3 个字节。因每次只能从存储器中取出一个字节，因此每次取指令需要 3 个时钟周期。此外，一个完整的指令周期需要四个时钟周期。所有这些变化使得控制信号变得更为复杂。

机器响应指令代码执行一系列操作称为执行指令，但这并不是说机器是有生命的东西，它也不是通过分析机器码来决定做什么。每一个机器码用唯一的方式触发各种控制信号，使机器产生各种操作。

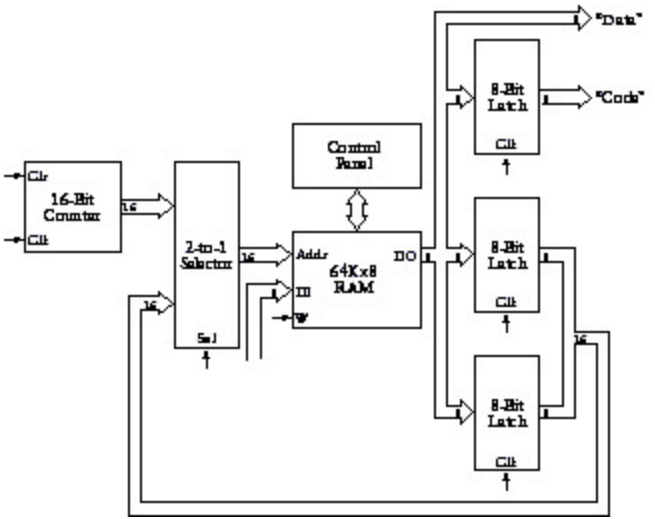
注意，为了使上述加法机更为有用，我们已经放慢了它的速度。利用同样的振荡器，它进行数字加法运算的速度只是本章列出的第一个自动加法器的 $1/4$ 。这符合一个叫作 TANSTAAFL 的工程原理，TANSTAAFL 的意思是“世界上没有免费的午餐”。通常，机器在某一方面好一点儿，在另一些方面必然会差一些。

如果不用继电器来建造这样一个机器，电路的大部分显然只是两个 64KB RAM 阵列。确实，早就该省去这些组件，并且一开始就决定只用 1KB 的存储器。如果能保证存储的所有东西都在地址 0000h~03FFh 之间，那么用少于 64kB 的存储器也能很好地解决问题。

然而，你可能也不会太在意用到了两个 RAM 阵列。事实上，也确实不用。前面介绍过的两个 RAM 阵列——一个存储代码，一个存储数据——使得自动加法器的体系结构变得尽可能清晰、简单。但既然已经决定每条指令占 3 个字节——用第二和第三个字节来表示数据的存储地址——就不再需要有两个独立的 RAM 阵列，代码和数据可存储在同一个 RAM 阵列中。

为了实现这个目标，需要一个 2-1 选择器来确定如何寻址 RAM 阵列。通常，像前面一样，其地址来自 16 位计数器。RAM 数据输出仍然连接到用来锁存指令代码及其 2 字节地址的三个锁存器

上，但它们的 16位地址是 2-1选择器的第二个输入。在地址被锁存后，选择器允许被锁存的地址作为 RAM阵列的地址输入：



数据

8 位锁存

器 代码

控制面板

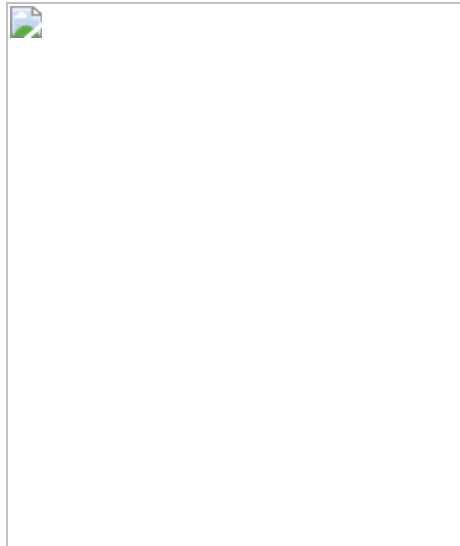
2-1 选

择器

8 位锁存器

8 位锁存器

我们已经取得了很大的进步。现在把指令和数据输入到一个 **RAM** 阵列中已成为可能。例如，下图显示出怎样把两个 **8**位数相加再减去第三个数：



把地址 **0010h** 处的字节装入累加器

把地址 **0011h** 处的字节加到累加器

从累加器减去地址 **0012h** 处的字节

把累加器中的内容存入地址 0013h 处

停止

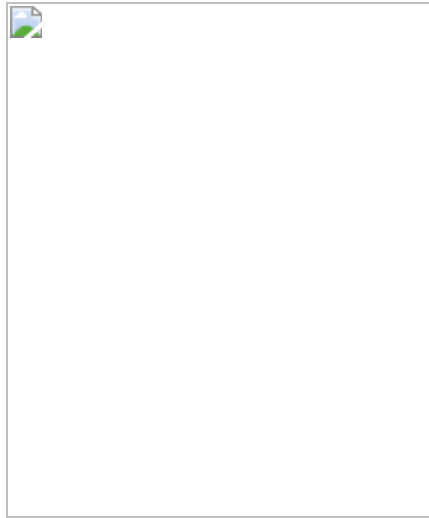
—— 最后结果放在这里

通常，指令开始于0000h，这是因为复位后计数器从 0000h处开始访问RAM阵列。最后的停止指令存储在地址000Ch处。可以把这3个数及其运算结果保存在RAM阵列中的任何位置（除了开始的13个字节，因为这些存储单元已经被指令占用），因而我们选择从0010h处开始存储数据。现在假设你需要再加两个数到结果中，你可以输入一些新的指令来替换你刚输入的所有指令，不过可能你并不想这样做。你也许更愿意在那些已有的指令末尾接着新的指令，但首

先得用一个新的装载指令来替换地址 000Ch 中的停止指令。此外，还需要两个新的加法指令、一个保存指令和一个新的停止指

令。唯一的问题在于有一些数据保存在地址 **0010h** 中，必须把这些数据移到更高的存储地址中，并且修改那些涉及到这些存储器地址的指令。

想一想，把代码和数据混放在一个 **RAM** 阵列中也许并不是一个迫切的问题，但可以肯定，这样的问题迟早会到来，因此必须解决它。在这种情况下，可能你更愿意做的就是从地址 **0020h** 处开始输入新指令，从地址 **0030h** 处开始输入新数据：



把地址 **0013h** 处的字节装入累加器

把地址 **0030h** 处的字节加到累加器

把地址 0031h 处的字节加到累加器

把累加器中的内容存入地址 0032h 处

停止

最后结果放在这里

注意第一条装载指令指向存储单元 0013h，即第一次运算结果存储的位置。因此现在有开始于0000h的一些指令、开始于0010h的一些数据、开始于0020h的另外一些指

令以及开始于0030h的另外一些数据。我们想让自动加法器从0000h处开始并执行所有的指令。我们必须从 000Ch处去掉停止指令，并用其他一些东西来替换它，但这样就足够了吗？问

题在于无论用什么来替换停止指令都会被解释为一个指令字节，并且至此存储器中每隔 3 个 字节——在000Fh、0012h、0015h、0018h、001Bh和001Eh处，字节也会被解释为一个指令字节。如果其中一个正好是 11h会怎样呢？这是一个保存指令。如果保存指令后的两个字节刚好 指向地址 0023h又会怎样呢？机器会把累加器的内容写入该地址中，但是该地址中已经包含有一些重要的东西。即使没有诸如此类的事情发生，加法器从存储器地址 001Eh的下一个地址中 取得的指令字节将在地址 0021h中，而不是0020h中，而0020h却正好是下一个指令的真实所在。

我们是否都同意不把停止指令从 000Ch处移走，而期待最佳方案呢？不过，我们可用一个叫作 **Jump**（转移）的新指令替换它。现在把它加入到指令表中。

操作码	代码
Load	10h
Store	11h
Add	20h
Subtract	21h
Add with Carry	22h

Subtract with Borrow 23h

Jump (转移) 30h

Halt FFh

通常，自动加法器顺序寻址 RAM 阵列。转移指令改变其寻址模式，而从 RAM 阵列的某个特定地址开始寻址。这样的命令有时也叫分支（branch）指令或者 goto 指令，即“转到另外一个地方”的意思。

在前面的例子中，可用转移指令来替换 000Ch 中的停止指令：

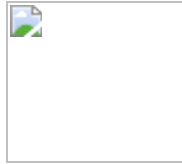


转移到地址 0020h 处的指令

30h 就是转移指令的代码，其下的 16 位地址表示自动加法器要读的下条指令的地址。因此，在前面的例子中，自动加法器仍从地址 0000h 处开始，执行一条装载、一条加法、

一条减法和一条保存指令，然后执行转移指令，接着继续从 0020h 处执行一条装载、两条加法和一条保存指令，最后执行停止指令。

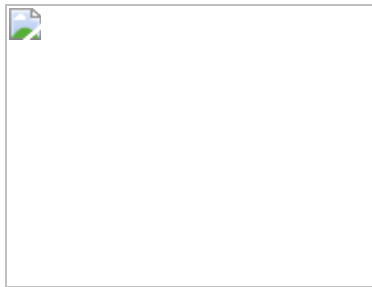
转移指令影响 16位计数器。当自动加法器遇到转移指令时，计数器被强制输入紧随转移指令代码的新地址，这可以通过组成 16位计数器的边沿触发的 D型触发器的预置（Pre）和清零(Clr)输入来实现：



前面曾讲过，正常操作下，预置和清零输入都应该为 0。但如果 $\text{Pre} = 1$ ，则 $Q = 1$ ；如果

$\text{Clr} = 1$ ，则 $Q = 0$ 。

如果想装载一个新值（称作 **A**，代表地址）到单个触发器中，可这样连线：



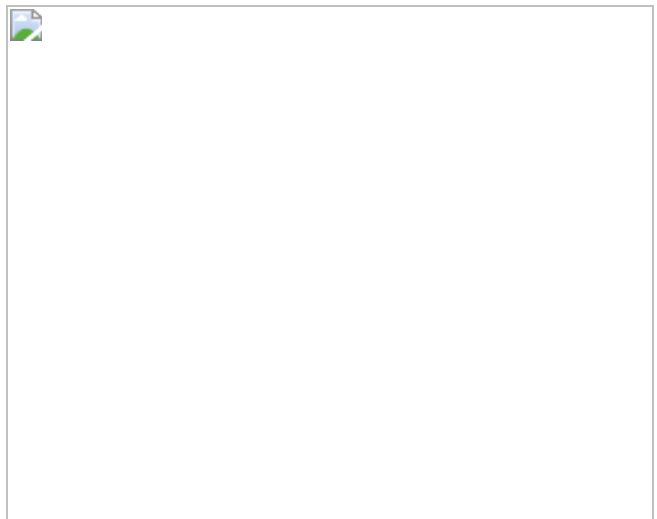
置位

复位

通常，置位信号为 0。这时，触发器的预置端为 0。除非复位信号为 1，否则清零端也为 0。这样触发器可以不通过置位信号就可以清零。当置位信号为 1 时，若 $A = 1$ ，则 $Pre = 1$ 且 $Clr = 0$ ；若 $A = 0$ ，则 $Pre = 0$ 且 $Clr = 1$ 。这意味着 Q 端的值设置为 A 端的值。

16 位计数器的每一位都需要一个这样的触发器。一旦装载一个特定的值，计数器将从那个值开始继续计数。

然而，这些变化并不大。从 RAM 阵列中锁存的 16 位地址既可作为 2-1 选择器（它允许该地址作为 RAM 阵列的地址输入）的输入也可作为 16 位计数器的输入并由置位信号设置：



置位 复位

16位计
数器

2-1 选

择器

控制面板

8 位

锁存 “代码”器

8 位锁存器

8 位 锁存 器

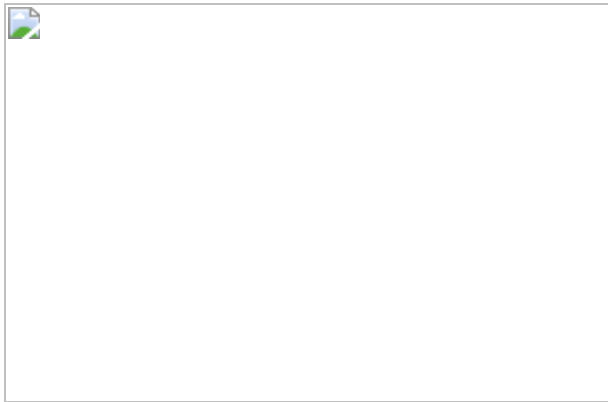
显而易见，只有当指令代码为 **30h**且其后面的地址被锁存，我们才必须保证置位信号为 **1**。

转移指令当然很有用，但它并非和一条只有时跳转而并非时刻跳转的指令一样有用，这样一个指令叫作条件转移。为了显示该命令如何有用，可提出这样一个问题：怎样才能让自动加法器完成两个 8 位数的相乘？例如，怎样才能得到像 A7h 乘以 1Ch 这样简单运算的结果？

很容易，不是吗？两个 8 位数相乘的结果是一个 16 位数。为了方便起见，乘法中的 3 个数都用 16 位数来表示。首要的工作是决定把乘数和乘积放在何处：



每个人都知道 A7h和1Ch（即十进制的 28）相乘的结果与 A7h相加28次的结果相同。因此，在1004h和1005h处的16位数就是累加结果。下图显示的是把 A7h加一次到那个位置的代码：



把地址 1005h 处的字节装入累加器

把地址 1004h 处的字节装入累加器

把地址 1001h 处的字节加到累加器

把 1000h 处的字节同进位 加到累加器

把累加器的内容存到 地址 **1005h** 处

把累加器的内容存入 地址 **1004h** 处

在这6条指令执行完后，存储单元 1004h和1005h处的16位数等于A7h乘以1。因此，为了使这16位数等于A7h乘以1Ch，这6个指令必须重复执行 27次。可以通过在地址 0012h 处接着输入 27次这6个指令来实现；也可以在 0012h处输入停止指令，然后按 28次复位键来得到最终结果。当然，这两个方案都不理想。它们需要你

做某些事情 (输入大批指令或者按复位键)的次数

和乘数相当。当然你不愿意这样去进行 16位数的乘法运算。

但是如果在 0012h 处输入转移指令会怎么样呢？这个指令使计数器从 0000h重新开始计数：



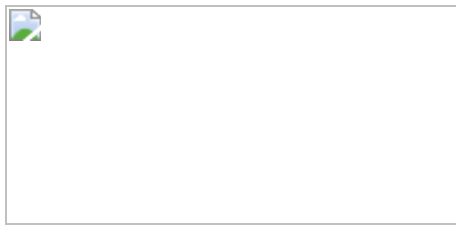
转移到地址 0000h 处的指令

这当然是一个技巧。第一次执行指令后，存储单元 1004h和1005h处的16位数等于 A7h乘1，然后转移指令使其返回到存储器顶部。第二次执行指令后，此 16位数等于 A7h乘2。终于，其

结果将等于 A7h乘1Ch。不过这样的过程并不会停止，它将不断地运行、运行、运行。我们想让转移指令做的是使循环过程只重复所需的次数，这就是条件转移，它实施起来

并不困难。我们要做的第一件事情就是增加一个与进位锁存器类似的 1位锁存器。因为只有 8

位加法器的输出全为 0时它才锁存 1，所以叫它零锁存器：



8 位加法器

零标志位

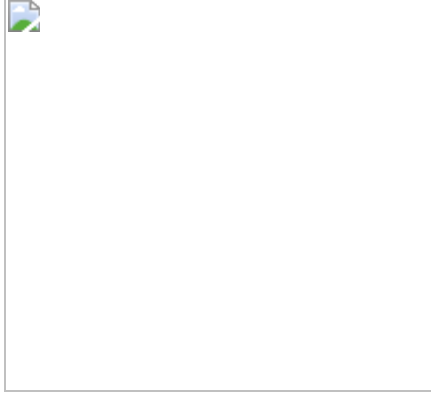
只有当或非门的 8 个输入全为 0 时，其输出才为 1。同进位锁存器的时钟输入一样，只有当 加法、减法、进位加法或借位减法指令运行时，零锁存器的时钟输入才锁存一个数，这个被 锁存的数值叫作零标志位。注意它，是因为它似乎行为相反：如果加法器输出全为 0，则零标志位为1；若加法器输出不全为 0，则零标志位为 0。

利用进位锁存器和零锁存器，可以在指令表中再添加四条指令：

操作码	代码
Load 10h	
Store 11h	
Add 20h	
Subtract 21h	
Add with Carry 22h	
Subtract with Borrow 23h	
Jump 30h	
Jump If Zero (零转移)	31h
Jump If Carry (进位转移)	32h
Jump If Not Zero (非零转移)	33h
Jump If Not Carry (无进位转移)	34h
Halt	FFh

例如，只有当零锁存器输出为 0 时，非零转移指令才转移到指定地址。换句话说，如果上一次加法、减法、进位加法和进位减法指令计算结果为 0，则没有转移发生。实现这个设计只 需在实现常规转移命令的控制信号上再加上一个控制信号：如果为非零转移指令，则只有当 零标志位为 0 时，16 位计数器的置位信号才被触发。

利用上述代码实现两个数的乘法所需的操作可由如下开始于地址 0012h 处的指令完成：



把地址 1003h 处的字节装入累加器

把地址 001Eh 处的字节加到累加器

把累加器的内容存到地址 1003h 处

若零标志位不为 0，转移到地址 0000h 处

停止

正如我们所设计的，循环一次后，位于 1004h 和 1005h 处的 16 位数等于 A7h 乘以 1。上图中 的这些指令把字节从 1003h 处装载到加法器中，此字节为 1Ch。再把这个字节与 001Eh 处的数据相加，此处数据正好是停止指令，但当然也是有效数字。把 FFh 同 1Ch 相加与从 1Ch 减去 1 的结果相同，都等于 1Bh。这个值不为 0，所以零标志位为 0，字节 1Bh 存回到地址 1003h 处。接下来是一条非零转移指令，零标志位没有置为 1，所以转移发生。下一条指令位于地址 0000h 处。

记住，存储指令不会影响零标志位。零标志位只能被加法、减法、进位加法、借位减法 指令所影响，因此它同这些指令中最近一个执行时所设置的值相同。

循环两次后，位于 1004h 和 1005h 处的 16 位数将等于 A7h 乘以 2。而 1Bh 加上 FFh 等于 1Ah，不是 0，因此又返回到存储器顶部。

循环到第 28 次时，位于 1004h 和 1005h 处的 16 位数等于 A7h 乘以 1Ch。位于 1003h 处的值等于 1，它将加上 FFh 结果等于 0，因此零标志位被置位。非零转移指令不再转移到存储器地址 0000h 处，相反，下一条指令为停止指令。至此，我们完成了全部工作。

现在可以肯定，很长一段时间以来我们已经装配了一组硬件，同时可以把它叫作计算机。当然，它只是一台原始的计算机，但它毕竟是一台计算机。它与我们以前设计的计算器的不同之处在于条件转移指令，控制重复或循环是计算机和计算器的区别。这里已经演示了条件转移指令是如何使得这台机器进行两个数的乘法运算的。用类似的方法，它也能计算两个数的除法。而且，还不局限于 8 位数。它能加、减、乘、除 16 位、24 位、32 位甚至更多位的数，而且如果它能实现这些操作，也就能计算平方根，对数和三角函数。

既然已装配了一台计算机，就可以开始使用一些计算机方面的词汇。我们装配的计算机归类为数字计算机，因为它采用的是离散值。曾经有过模拟计算机，

但它们正逐渐消失。（数字数据是离散数据，是具体的确定的值；而模拟信息是连续的、在整个范围内变化的值。）

数字计算机有 4 个主要部分：处理器、存储器、至少一个输入设备和一个输出设备。上述机器中，存储器是一个 64KB 的 RAM 阵列。输入和输出设备分别是 RAM 阵列控制面板上的几行开关和灯泡。这些开关和灯泡使人们可以输入数据到存储器并检查结果。

处理器是计算机中除存储器、输入/输出设备以外的一切东西。处理器也叫中央处理器单元或 CPU。再通俗一点儿，处理器有时也称作计算机的大脑。但尽量避免用这样的术语，这是因为在本章中我们所设计的东西根本不像大脑。（今天，微处理器这个词用得非常普及。微处理器只是一个很小的处理器，通过采用第 18 章将要讲到的技术而实现。但此刻我们用继电器所建造的东西则很难用“微”来定义。）

我们所建造的处理器是一个 8 位处理器。累加器宽度为 8 位，并且许多数据通路的宽度都是 8 位，只有 RAM 阵列的地址的数据通路是 16 位的。如果用 8 位的地址通路，则存储器容量只能限于 256 字节而非 65 536 字节，那样处理器则有太大的局限性。

处理器有一些组件。已经确定的一个是累加器，它是一个简单的锁存器，用来在处理器内部保存数据。我们所设计的计算机中，8 位反向器和 8 位加法器一起称作算术逻辑单元或 ALU。ALU 只能进行算术运算，主要是加法和减法。在稍微复杂一点儿的计算机中（我们将会看到），ALU 也可进行逻辑运算，如“与”、“或”、“异或”。16 位计数器叫作程序计数器 PC。

我们的计算机是用继电器、电线、开关和灯泡建造的，所有这些硬件。与之对应，

指令和输入存储器中的其他数据叫作软件，之所以叫“软件”是因为它们比硬件更容易改变。当谈论计算机时，“软件”和“计算机程序”，更简单地讲“程序”是同义的，编写软件也称作计算机程序设计。当采用一系列计算机指令使计算机进行两个数的乘法时，我们所做

的工作就是计算机程序设计。通常，在计算机程序中，可以区分代码（即指令）和供代码使用的数据。有时这种区分

并不明显，如停止指令还可作为数—1执行双重功能。计算机程序设计有时也叫编写代码或编码。有时计算机程序员也叫编码员，尽管一些人

可能认为这是一个贬义的名词。程序员更愿意被称作“软件工程师”。处理器可以响应的操作码（如指装载和存储的10h和11h）叫作机器码，或机器语言。之

所以用“语言”这个术语是因为机器码类似于可读/写的人类语言可被机器理解和响应。我们要用很长的短语表示机器所执行的指令，如：进位加法（Add with Carry）。通常，

机器码都分配指定了用大写字母表示的短的助记符，这些助记符有2或3个字符。下面是一系列可能的上述计算机所能识别的机器码的助记符：

操作码	代码	助记符
	10h	LOD
装载（Load）		
	11h	STO

保存 (Store)

加 (Add)

20h

ADD

减 (Subtract)

21h

SUB

进位加 (Add with Carry)

22h

ADC

借位减 (Subtract with Borrow)

23h

SBB

转移 (Jump)

30h

JMP

零转移 (Jump If Zero)

31h

JZ

进位转移 (Jump If Carry)

32h

JC

非零转移 (Jump If Not Zero)

33h

JNZ

无进位转移 (Jump If Not Carry)

34h

JNC

停止 (Halt)

FFh

HLT

这些助记符特别适于和另外一对简洁短语结合使用。例如，不说像“把 1003h处的值装载 到累加器中”这样罗嗦的话，而是用下面语句来代替：

L0D A , [1003h]

位于助记符 **LOD** 右边的 **A** 和 **[1003]** 叫作操作数，它们是特定的装载（**Load**）指令的操作对象。左边的操作数为目的操作数（**A** 代表累加器），右边的为源操作数，方括号表示要装载到累加器中的值不是 **1003h**，而是存储在存储器地址 **1003h** 中的值。

同样，指令“把 **001Eh** 处的字节加到累加器中”可简写为：

```
ADD     A, [001Eh]
```

而“把累加器中的内容保存到地址 **1003h** 处”记作：

```
STO     [1003h], A
```

注意，目的操作数（存储指令的存储单元）仍然在左边，源操作数在右边。累加器的内容存储在地址 **1003h** 处。指令“若零标志位不为 1 则转移到 **0000h** 处”可简洁地记作：

```
JNZ 0000h
```

该指令中没有使用方括号，这是因为该指令是转移到地址 **0000h** 处而不是转移到地址

0000h中保存的值所表示的位置处。用缩写指令的形式来表示很方便，因为指令能以可读的方式连续列出来而不需画出存储

器的分配图。为了表示某一指令存储在某一地址，可以用一个十六进制地址后加冒号来表示，如下所示：

```
0000h: LOD A, [1005h]
```

下面表示了一些存储在某一地址的数据：

```
1000h: 00h, A7h
```

```
1002h: 00h, 1Ch
```

```
1004h: 00h, 00h
```

用逗号隔开的两个字节表示第一个字节保存在左边的地址中，第二个字节保存在紧接着该地址的下一个地址中。上述三行相当于：

```
1000h: 00h, A7h, 00h, 1Ch, 00h, 00h 因此，整个乘法程序可写成如下一系列语句：  
0000h: LOD A, [1005h]
```

```
ADD A, [1001h]
```

```
STO [1005h], A
```

```
LOD A, [1004h] ADC A, [1000h] STO [1004h], A
```

```
L0D A, [1003h] ADD A, [001Eh] ST0 [1003h], A
```


001Eh: HLT

JNZ 0000h

1000h: 00h, A7h

1002h: 00h, 1Ch

1004h: 00h, 00h

使用空格和空行只是为了使程序具有更好的可读性，以方便人们阅读程序。写代码时最好不要用真实的数字地址，因为它们是会变的。例如，如果要把数字存储到

地址 2000h~2005h处，需要重写许多语句。较好的方法是使用标号来指定存储单元，这些标号是简单的单词，或类似于单词的东西，如：

```
BEGIN:  LOD A, [RESULT+1]
```

```
        ADD A, [NUM1+1] STO [RESULT+1], A
```

```
        LOD A, [RESULT] ADC A, [NUM1] STO [RESULT], A
```

```
        LOD A, [NUM2+1] ADD A, [NEG1]
```

```
STO [NUM2+1], A
```

```
JNZ BEGIN
```

```
NEG1:      HLT
```

```
NUM1:  00h,  A7h NUM2:  00h,  1Ch RESULT:  00h,  00h
```

注意，标号 NUM1、NUM2和RESULT都表示保存两个字节的存储单元。在这些语句中，标号 NUM1+1、NUM2+1和RESULT+1都指向特定标号后的第二个字节。注意，*NEG1*

(negative one) 用来标记 HLT指令。此外，为了不忘记这些语句的意思，可以加上一些注释，它们与语句之间用分号隔开：

```
BEGIN:  LOD A,  [RESULT+1]
```

```
ADD A,  [NUM1+1]      ; Add low-order byte (加低字节)  STO [RESULT+1], A
```

```
LOD A,  [RESULT]
```

```
ADC A,  [NUM1]        ; Add high-order byte (加高字节)  STO [RESULT], A
```

```
LOD A,  [NUM2+1]
```

```
ADD A,  [NEG1]        ; Decrement second number (第二个数减 1)  STO [NUM2+1], A
```

```
JNZ BEGIN

NEG1:  HLT

NUM1:  00h,  A7h NUM2:  00h,  1Ch RESULT:  00h,  00h
```

以上表示的是一种计算机程序设计语言，称作汇编语言。它是全数字的机器代码和指令描述性语言的综合，且存储器地址用符号表示。人们有时会把机器语言和汇编语言弄混淆，因为它们表示同种事情的两种不同的方法。汇编语言的每条语句都对应于机器代码的特定字节。

如果你想为本章所创建的计算机编写程序，你可能首先想用汇编语言写出来（在纸上）。然后，在认为它正确并准备测试它时，可以对它进行手工汇编：这意味着用手工的方法把每一个汇编语句转换成机器代码，仍然写在纸上。接着，你可以用开关把机器码输入到 **RAM**阵列并运行该程序，即让机器执行指令。

学习计算机程序设计的概念时，不可能很快就能正确知道程序的毛病所在。编写代码时

——特别是用机器代码——很容易产生错误。输入一个错误的数字已经很不好，但如果输入一条错误的指令会怎么样呢？本想输入 **10h**(装载指令)，但却输入了 **11h**（保存指令），不但机器不会把期望的数据装载，而且该处的数据还会被累加器中的内容覆盖。

一些错误可以导致难以预料的结果。假设使用无条件转移指令转移到没有有效指令代码的位置，或者偶然使用存储指令覆盖了一些指令，任何事情都可能发生（经常如此）。

上述乘法程序中也有一些毛病。如果你执行它两次，则第二次将会是 **A7h**乘以256，并且结果将加到原来计算的结果中。这是因为程序执行一次后，地址 **1003h**处的值为 **0**。当程序第二次执行时，**FFh**将加到那个值中，其结果不为 **0**，程序将继续执行直到它为 **0**。

我们已看到上述机器可以进行乘法运算，同样，它也可以进行除法运算。此外，它可利用这些基本功能进行平方根、对数和三角函数的计算。机器所需要的只是用来进行加法、减法的硬件及利用条件转移指令来执行适当代码的一些方法。正如一个程序员所说：“我可以用软件完成其余功能”

当然，软件可能相当复杂。许多书中都描述了一些算法供程序员解决专门的问题，本书还没准备这样做。我们一直在考虑自然数而没有考虑如何在计算机中表示十进制小数，我们将在第23章介绍它。

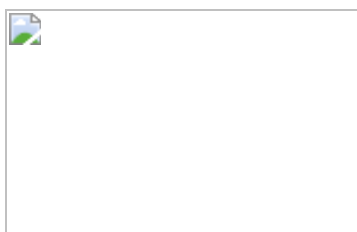
前面已说过几次，建造这些设备的所有硬件在 100多年前就有了。但本章中出现的计算机在那时却没有建造出来。在 20世纪 30年代中期，最早的继电器计算机制造出来时，包含在设计中的许多概念还未形成，直到 1945年左右人们才开始意识到。例如，直到那时候，人们仍然设法在计算机内部使用十进制数而不是二进制数；计算机程序也并非总是存储在存储器中，而是有时把它存在纸带上。特别是早期计算机的存储器非常昂贵且体积庞大。不管是在 100年前还是在现在，用 500万个电报继电器来建造 64KB的RAM阵列都是荒唐的。

当我们展望和回顾计算器和计算装置的历史时，可能会发现根本没必要建造这样精致的继电器计算机。就像在第 12章提到的，继电器最终会被真空管和晶体管这样的电子设备所取代。或许我们也会发现他人制造的相当于我们设计的处理器和存储器的东西能小到放在手掌中。

第 18 章 从算盘到芯片

纵观历史，人类发明了很多灵巧的工具和机器以满足广泛的需求，从而使数学运算变得更容易了些。虽然人类天生就有使用数字的能力，但仍能经常需要帮助。人们常遇到一些自己不能轻易解决的问题。

数字可看成是早期帮助人类记录商品和财富的工具。许多文明，包括古希腊和美洲土著，都借用石子或谷物来计数。在欧洲使用计数板，而在中国则对由框和珠子组成的算盘较为熟悉：



没有人真的喜欢乘法和除法，但却有人为它做过什么，苏格兰数学家 John Napier(1550- 1617)就是这少数人中的一个。他发明了对数来简化这些操作，两数之积简化为它们对数的和。因此，如果你想使两数相乘，先在对数表中分别查出它们的值，然后相加，再用相反的方法查对数表就可得到它们的积。

对数表的建立，使得随后 400年里一些最伟大的思想家一直为此忙碌，而另一些人却在设计使用小装置来代替对数表。一种有对数标尺的滑尺已有很长的历史了，它由 Edmund Gunter

(1581-1626) 发明并由 William Oughtred (1574-1660) 修正。1976年, 当Keuffel&Esser公司 将其公司最后制造的滑尺捐赠给华盛顿特区的 Smithsonian 学院时, 滑尺的历史也就宣告结束了, 其中的原因是手持计算器的出现。

Napier也发明了一种乘法辅助器, 它由刻在骨头、号角、象牙上的数字条组成, 因而这样的辅助器称为 Napier骨架。1620年左右, Wilhelm Schickard (1592-1635) 制造出了最早的有点儿自动功能的由 Napier骨架组成的机械计算器。几乎在同时出现了由互相连结的轮子、齿轮和水平仪组成的另外一种计算器, 这种机械计算器的两个最主要的制造者是数学家和哲学家布莱兹·帕斯卡 (1623-1662)和莱布尼兹 (1646-1716)。

你一定能记得最初的 8位加法器和能自动进行多于 8位数的加法计算的计算机中的进位是多么令人讨厌。进位原先似乎只是加法运算中的一个小问题, 但在加法机中却成了一个中心问题。即使设计一个能进行除进位外的所有工作的加法机, 也不能说工作就算完成了。

进位处理是否成功是评估老式计算机的关键。例如, 帕斯卡设计的进位机制禁止减法运算。为了进行减法, 必须加上 9的补码, 这在第 13章中已经讲到。直到 19世纪后期, 才出现了真正可以为人们所使用的机械计算器。

一个奇特的发明对计算的历史产生了深远的影响，就像它对纺织所产生的深远影响一样，这就是约瑟夫·玛丽·杰奎德 (1752-1834) 所发明的自动织布机。杰奎德织布机（大约产生于 1801年）使用上面已打孔的金属卡片（就像钢琴上的金属卡片）来控制编织物的图案。杰奎德的一大杰作就是用黑白丝线织成的自画像，为此使用了大约 1万张卡片。

在18世纪（甚至直到 20世纪40年代），计算机就像一个以计算数字谋生的人。使用星星进行航海导航经常需要对数表，并且三角函数表也是必需的。如果需要发布新表，则需要许多计算机来工作，然后把结果汇总起来。当然，在这一过程的任何阶段，即从初始化计算到设置类型来打印最后几页都可能会出现错误。



从数学表中消除错误的愿望激发了查尔斯·巴贝芝（1791—1871）。巴贝芝是一位英国的数学家和经济学家，他和摩尔斯差不多是同一时代的人。

在那时，数学表（以对数表为例）并不是通过计算表中每一项确切的对数值而建立的，因为这得花费很多时间。取而代之的是选择一些数进行对数计算，而介于这些数中间的那些数则采用插补，即称作差分的方法，通过相对简单的计算来得到。大约在1820年，巴贝芝认为可以设计并制造一台机器来自动建立表，甚至可以到自动设置打印类型这一步，这样可以消除错误。他构想

了差分机，这是一个很大的机械加法机。通过 切换，可使位于 10 个不同位置的轮子来表示各位数的十进制数

字，负数用 10 的补码来计算。尽管一些早期的模型可以证明巴 贝 芝的设计是可行的，并且也从英国政府获得了一些支持，但差分机却从未完成过。巴贝芝 于1833年放弃了这一工作。

然而，就在那个时候，巴贝芝又有了一个更好的构想，这就是解析机 (重复的设计和再设计不断耗费着巴贝芝的生命，直到他死去)，解析机是 19 世纪最接近计算机的发明。在巴贝芝 的设计中，有一个存储系统（类似于今天存储器的概念）和运算器（算术单元）。乘法由重复 加法来实现，除法由重复减法来实现。

解析机最精华的部分在于它可以用卡片来编程，这些卡片是由杰奎德的按图案编织的织 布机上的卡片经过改造而制成的。正如艾达·奥古斯塔，即拉弗雷斯女伯爵（ 1815-1852）在 她翻译的由一个意大利数学家写的，关于巴贝芝解析机的文章的按语里写的：“我们可以说解 析机编织的是代数模型，正如杰奎德织布机编织的是花和叶一样”。

巴贝芝可能是第一个意识到计算机中条件转移的重要性的人。拉弗雷斯女伯爵关于此也 曾写道：“操作循环必须理解成一批操作，这些操作可以重复多次。无论是只重复两次还是无 穷次，都是一个循环，归根到底重复组成了循环的这些操作。许多情况下，还会出现一个或 多个循环的重复，即循环的循环或多个循环的循环”。

尽管差分机最终由 Georg和Edvard Scheutz 父子在 1853年制成，但巴贝芝的机器那时已被 遗忘了很久，直到 20世纪30年代人们开始追寻 20世纪计算机的根源时才再次想起。巴贝芝曾 经做的东西已经被后来的技术所超越，除了他对自动化的超前认识外，他并没有为 20世纪的 计算机工程留下什么东西。

计算机历史上另一个里程碑来源于美国宪法第二部分的第一篇。这一部分里除其他事情 外还要求每 10年进行一次人口普查。1880年人口普查的时候，人口信息按年龄、性别及祖籍

来收集，数据的收集差不多花了七年的时间来进行。

由于担心 1890年的人口普查可能需要超过 10年的时间，人口普查局寻求使该系统工作自

动化的可能性并选用了赫曼·霍勒瑞斯 (1860—1929)开发的机器，此人是 1880年人口普查的



统计员。 5 1

。

。

霍勒瑞斯的想法是采用马尼拉穿孔卡片，大小是 68 □□34

（虽然霍勒瑞斯不可能知道巴贝芝如何使用卡片在他的解析机上编

程，但他却很熟悉杰奎德织布机上卡片的使用。）卡片上的孔组成 24 列，每列 12 个，这样共有 288 个位置，这些位置表示某个人在人口普查记录中的某些特征。普查员通过在卡片的适当位置上打 1/4 英寸的方孔来标识这些特征。

本书可能使得人们习惯于用二进制码的概念来思考问题，因此，你可能马上会想到卡片上的 288 个穿孔点可以存储 288 位信息，但是，这些卡片并不是这样用的。

例如，在纯二进制系统中，人口普查卡片会有一个位置来表示性别，可以用打孔表示男性，未打孔表示女性（或者相反）。但是，霍勒瑞斯的卡片用两个位置表示性别，一个位置打孔表示男性，另一个位置打孔表示女性。同样，用两个穿孔表示年龄，一个穿孔指明一个 5 年的年龄范围：0~4、5~9、10~14 等等，另一个孔用 5 个位置中的一个来表明在该范围内的确切年龄。年龄编码需要卡片上总共 28 个位置。而纯二进制系统只需要 7 个位置就可编码 0~127 的任何年龄。

我们应该原谅霍勒瑞斯在记录人口普查信息时没有采用二进制系统，对于 1890年的人口普查员来说，把年龄转换成二进制数要求太高了一些。还有一个实际原因来解释穿孔卡片系统为什么不能全部是二进制的是因为二进制系统可能出现这样一种情形，即所有的孔都被打孔，使得卡片很脆弱，结构不牢固。

人口普查的数据收集可进行统计或制成表格。你可能想知道每一个区域内有多少人生活，当然，你也可能对人口的年龄分布统计信息感兴趣。正因为如此，霍勒瑞斯制造了一个制表机，它能结合手工操作和自动操作。操作员把一个有 288个弹簧针的板子压到每一个卡片上，对应于卡片上每一个穿孔的针接触水银池形成电路，电路触发电磁铁使十进制计数器计数。

霍勒瑞斯在分类卡片的机器上也用了电磁铁。例如，如果需要统计所记录的每一个职业的年龄资料，首先需要按职业对卡片分类，然后对每一个职业统计年龄资料。分类机与制表机一样用手压，但分类机使用电磁铁打开一个开口，对应于 26个分隔区域中的一个，操作员把卡片放入分隔区域，然后用手工关上开口。

这项实验在自动进行 1890年的人口普查工作中取得了巨大成功，处理了超过 6200万张的卡片，包含的数据是 1880年人口普查的 2倍，而数据处理只花了大约 1880年人口普查所花时间的1/3。霍勒瑞斯和他的发明享誉全球。1895年，他甚至到了莫斯科并成功地卖出了他的设备，该设备在 1897年第一次用于俄罗斯的人口普查。

霍勒瑞斯开始进行各种活动。1896年，他创立了制表机公司，出租和出售穿孔卡片设备。1911年，经过合并，该公司成为计算-制表-记录 (computing-Tabulating-Recording) 公司，即 C-T-R 公司。到 1915年，C-T-R的主席是 Thomas J.Watson(1874-1956)，他在 1924年把公司的名字改为国际商用机器公司，即 IBM。

1928年，原先的 1890年的人口普查卡片已经演化成为著名的“不会卷曲、折叠、翘页”的IBM卡片，有 80列12行。它们用了 50多年，即使在最后几年也被称作霍勒瑞斯卡片。在第 20、21和24章将要讲到这些卡片的影响。

在把目光移到 20世纪之前，不要对 19世纪那个年代有太多的偏见。显然，本书主要着眼于数字系统的发明，这些发明包括电报、布莱叶盲文、巴贝芝机器和霍勒瑞斯卡片。当与数字概念和数字设备一起工作时，很容易会把整个世界都想像成数字世界。但是，19世纪的特征更多体现在那些不是数字的发明及发现上。的确，我们感受到的自然世界只有很小一部分是数字的，它更接近于是连续的而不那么容易被量化。

尽管霍勒瑞斯在他的卡片制表机和卡片分类机上用了继电器，但是直到 20世纪30年代中期，人们才真正开始用继电器来制造计算机（后来叫机电式计算机）。这些机器上用的继电器通常不是电报继电器，而是那些用来在电话系统中控制呼叫路由的继电器。

早期的继电器计算机并不像我们在上一章中制造的继电器计算机（将会看到，后者计算机的设计基础是基于从 20世纪70年代开始的微处理器）。特别地，今天对我们来说计算机内采用二进制数是显然的，但那时并不是这样。

我们所设计的继电器计算机与早期的继电器计算机之间的另一个区别是在 20世纪30年代，没有人会狂热到用继电器构造 524 288位的存储器！所需的造价、空间及功耗使得这样大的存储器不可能实现。可用的很小的存储器只是用来存储中间结果，而程序本身存储在像带有穿孔的纸带这样的物理媒体上。的确，把代码和数据放入存储器的处理方式是一个很现代化的概念。

按年代排列，第一个继电器计算机似乎是由 Conrad Zuse（1910-1995）建造的。1935年 当他还是一个工程系的学生的时候，他就开始在他父母位于柏林的住所里制造计算机。他采用的是二进制

数，但却是早期版本，且用的是机械存储器而不是继电器。Zuse 在老式的 35 毫米的电影胶片上穿孔来进行计算机编程。

1937 年，贝尔电话实验室的 George Stibitz（1904-1995）把一对电话继电器安装在家里，并且又连接了一个 1 位加法器到餐桌上，后来他的夫人称它为“K 机器”（K 表示 kitchen，厨房）。该实验导致在 1939 年产生了贝尔实验室的复数计算机。

与此同时，哈佛大学的研究生 Howard Aiken（1900-1973）因需要某种方法来做大量重复计算，从而使得哈佛大学和 IBM 合作，制造出了最终称为 Harvard Mark I 的自动顺序控制计算机（ASCC：automated sequence controlled calculator），此项工作在 1943 年完成。这是第一台打制表格的数字计算机，它终于实现了巴贝芝的梦想。Mark II 是以巨大的继电器为基础的机器，使用了 13 000 个继电器。由 Aiken 领导的哈佛计算实验室讲授了计算机科学的第 1 课。

继电器并不是制造计算机的最好器件，因为它是机械的，工作时需弯曲一个金属簧片，如果超负荷工作，簧片就会折断；如果有一小片污垢或纸片粘在触点之间，继电器就会失效。一个著名的事件发生在 1947 年，从 Harvard Mark II 计算机的一个继电器中找到一只蛾子。Grace Murry Hopper（1906—1992）1944 年加入 Aiken 的小组，此人后来在计算机程序设计语言领域非常有名。他在计算机日志中记录了这只蛾子，写道“第一次发现了真正的 bug”。

继电器的一种可能的替代品是真空管，真空管由 John Ambrose Fleming（1849—1945）和 Lee de Forest（1873—1961）发明用来同无线电设备连接。到 20 世纪 40 年代，真空管早已用来放大电话信号。事实上，每一家的落地式收音机都装上了用来放大无线电信号的真空管，

以便人们能听见。真空管可以连接成与、或、与非和非门，这一点非常像继电器。逻辑门是由继电器还是由真空管来制造的并不重要。利用逻辑门可集成加法器、选择器、

译码器、触发器和计数器。前面几章讲的基于继电器的器件在当继电器被换成真空管时仍然可用。

不过真空管也有问题，它们昂贵、耗电量大、散发的热量多。然而最大的问题在于它们最终会被烧毁，这也就是它们的寿命问题。有真空管收音机的人就习惯于隔一段时间更换这些管子。电话系统设计成有许多多余的管子，因此损失点儿管子也不是大的问题。（没有人能指望电话系统不出一点儿问题。）然而计算机中的一个管子烧毁以后，并不能很快被检测到，而且，计算机中使用了如此多的真空管，可能每几分钟就会烧毁一个。

使用真空管相对于继电器的最大好处在于它每百万分之一秒（即1微秒）就可以跳变一次。真空管改变状态（开关闭合或断开）的速度比继电器快1000倍，在最好的情况下，继电器状态的变化大约需1毫秒，即千分之一秒。有趣的是，在早期计算机的研究中，速度问题并不是最重要的，这是因为早期计算机总的计算速度与机器从纸带或电影胶片中读取程序的速度密切相关。正是因为计算机是基于这种方式制造的，真空管比继电器速度快多少也就无关紧要了。

在20世纪40年代初，真空管开始在新的计算机中替换继电器。直到1945年，晶体管制成。正如继电器机器称为机电式计算机，真空管则是第一台电子计算机的基础。

在英国，Colossus计算机（1943年开始使用）用于破译德国的“Enigma”代码生成器生成的密码。为这个项目（和英国以后的一些计算机项目）做出贡献的人是艾伦·M·图灵（1912—1954），他当时由于写了两篇很有影响的论文而闻名于世。第一篇论文发表于1937年，其中首先提出了“计算能力”的概念，用以分析计算

机可以做到和不能做到的事。他构思出了现在称为图灵机的计算机抽象模型。图灵写的第二篇著名论文的主题是人工智能，他介绍了一个测试机器智能的方法，现在称作图灵测试法。

在摩尔电气工程学校， J.Presper Eckert(1919 -1995) 和 John Mauchly(1907—1980) 设计了 ENIAC （ electronic numerical integrator and computer， 电子数字积分器和计算机）。它采用了 18 000个真空管，于 1945年末完成。纯粹按吨位（大约 30吨）计算， ENIAC是曾经制造出来的（也许以后也是）最大的计算机。到 1977年，你可以在电器行买到更快的计算机。然而， Eckert和Mauchly的专利却被 John V.Atanasoff(1903—1995)给阻挠了。 Atanasoff在早期曾设计了一个电子计算机，但它从未很好地工作过。



ENIAC引起了数学家约翰·冯·诺依曼 (1903—1957) 的兴趣。从1930年开始，匈牙利出生的冯·诺依曼就一直住在美国。他是一个令人瞩目的人物，因能在脑子里构思复杂的算法而享有很高的声誉，他是普林斯顿高级研究院的一名数学教授，研究范围很广，从量子理论到游戏理论的应用再到经济学。

冯·诺依曼帮助设计了 ENIAC 的后继产品 EDVAC

(electronic discrete variable automatic computer)。特别是在 1946 年与 Arthur W .Burks和Herman H.Goldstine 合写的论文

《Preliminary Discussion of the logical Design of an Electronic Computing instrument》中，他描述了有关计算机的几点功能 这些功能使得 EDVAC比ENIAC更先进。EDVAC的设计者感觉

到在计算机内部应该使用二进制数，而 ENIAC用的是十进制数；计算机应该具有尽可能大的 存储器，当程序执行时，这个存储器可用来存储程序代码和数据；（ ENIAC中的情况不是这样，对 ENIAC进行编程是通过断开开关和插上电缆来进行的。）指令应该在存储器中顺序存放 并用程序计数器来寻址，但也应该允许条件转移。这种设计思想叫作存储程序概念。

这种设计思想是重要的革命化的一步，今天称为冯·诺依曼体系结构，上一章建造的计算机就是典型的冯·诺依曼机器。但冯·诺依曼体系结构也带来冯·诺依曼瓶颈，冯·诺依曼型机器需要花费大量的时间从存储器中取出指令来准备执行。应该还记得第 17章最后设计的计算机取指令的时间占整个指令周期的 3/4。

在EDVAC时代，用真空管构建存储器是不值得的，因而人们使用一些古怪的方法来解决这个问题。一个成功的方法就是水银延迟线存储器，它使用 5英尺长的水银管子。在管子的一端，每隔1微秒向水银发一个小脉冲。这些脉冲需要 1毫秒的时间到达管子的另一端（此时，它们像声波一样会被检测到，然后送回到开始的地方），因此每个水银管可存储大约 1024位信息。直到20世纪 50年代中期，磁芯存储器才开发出来。这种存储器由大量的环绕着电线的电磁金属环组成，每个小环保存 1位信息。在磁芯存储器被其他技术取代后的相当一段时期内，

还经常听到老程序员把由处理器访问的存储器称作磁芯。在20世纪40年代，冯·诺依曼并不是唯一一个对计算机的本质进行概念上思考的人。克劳德·香农 (1916年出生)是另外一个有着重大影响的思想家。第 11章曾经提到他 1938年

的硕士论文，论文中确立了开关、继电器和布尔代数之间的关系。1948年，当他在贝尔电话实验室工作时，他在《 Bell System Technical Journal 》上发表了一篇题为《 A Mathematical Theory of Communication 》的论文，其中不仅引入了“位”的概念，而且确立了一个现代称为“信息理论”的研究领域。信息理论

涉及在噪声（经常阻碍信息传送）存在的情况下传送 数字信息以及如何进行信息补偿等问题。1949年，他写了第 1篇关于编写让计算机下棋的程序 的文章；1952年他设计了通过继电器控制的机械老鼠，这个老鼠可以在迷宫中记住路径。香农同时也因为他会骑独轮车，玩变戏法而在贝尔实验室很出名。

Norbert Wiener(1894-1964)18岁时就在哈佛大学取得了数学博士学位，因《Cybernetics, or Control and Communication in the Animal and Machine》（1948）一书而闻名于世。他首次使用控制论（Cybernetics）这个词来表示一种把人及动物的生物活动与计算机及机器人的机理联系起来的理论。在现代文化里，广泛使用 cyber-前缀表示与计算机相关的东西。更特别的是，成千上万的计算机通过因特网进行的互连称作 cyberspace（信息空间），这个词来自科幻小说作家 William Gibson 1984 年的小说《Neuromancer》中的词 cyberpunk。

1948年，Eckert-Mauchly计算机公司（Remington Rand公司的后继者）开始开发第一台商用计算机 UNIVAC(universal automatic computer)，并于 1951年完成。第一台被送往人口普查局。UNIVAC的首次网络应用是用于 CBS，用来预测 1952年的总统选举结果。Walter Cronkite 称它为“电脑”。同样是在 1952年，IBM 发布了它的第一个商用计算机系统，即 701。

从此，开始了社团和政府使用计算机的漫长历史。然而，之所以对这段历史感兴趣可能是因为我们要追踪另一段历史轨迹——即降低计算机造价和大小并且使它进入家庭的轨迹，它开始于 1947 年一场几乎不被人注意的电子技术突破。

贝尔电话实验室许多年里都是这样一个地方：聪明的人可以在此做他感兴趣的任何事。

所幸的是，他们之中有人对计算机感兴趣，如已经提到的 George Stibitz 和 Claude Shannon，

他们在贝尔实验室工作的时候都为早期的计算机作出了突出的贡献。后来，在 20 世纪 70 年代，贝尔实验室诞生了很有影响的操作系统 UNIX 和程序设计语言 C 语言，这些在随后的几章里将要讲到。

1925 年 1 月 1 日，美国电话电报公司正式把它的科学和技术研究部分与商业部分分离，另外建立附属机构，这样贝尔实验室诞生了。贝尔实验室的主要目的是为了研究能够提高电话系统性能的技术。幸运的是，它的要求非常含糊，可以包含所有的事情。但在电话系统中，一个明确的长期的目标是：在线路上传输的声音信号能不失真地放大。

从 1912 年起，贝尔电话系统就采用了真空管放大器，大量的研究和工程人员着手提高电话系统使用的真空管的性能。尽管这样，真空管仍然有许多问题。管子体积大，功耗大且最终会烧毁。不过，在当时却是唯一的选择。

1947 年 12 月 16 日，当贝尔实验室的两个物理学家 John Bardeen(1908—1991)和 Walter Brattain(1902—1987)在装配一个不同类型的放大器时，所有的一切都改变了。这种新型放大

器由锗片——一种称作半导体的元素——和一条金箔构成。一个星期后，他们给他们的上司 William Shockley (1910—1989) 进行了演示。这就是第一个晶体管，一种被人们称为 20 世纪最伟大的发明的器件。

晶体管不是凭空产生的。8 年前，即 1939 年 12 月 29 日，Shockley 在笔记本写下：“今天我想用半导体而不是真空管做放大器在原理上是可能的。”第一个晶体管被发明以后，随后许多

年它继续被完善。1956 年，Shockley、Bardeen 和 Brattain 获得诺贝尔物理学奖——“因为他们 在半导体上的研究并且发明了晶体管。”

本书的前面谈到了导体和绝缘体。之所以称作导体是因为它们非常容易导电，铜、银和金都是很好的导体。并非巧合，所有这三种元素都在元素周期表的同一列。

前面讲过，原子中的电子分布围绕在原子核的外层。上述三种导体的特征是只有一个单

独的电子在最外层。这个电子很容易与原子的其余部分分离并自由移动形成电流。与导体相对的是绝缘体，像橡皮和塑料，它们几乎不能导电。

锗和硅元素（还有一些化合物）称为半导体，并不是因为它们的导电性是导体的一半，而是因为它们的导电性可以用多种方法来控制。半导体最外层有 4 个电子，是最外层所能容纳电子最大数目的一半。在纯半导体中，原子彼此非常稳固地结合在一起，具有与金刚石相似

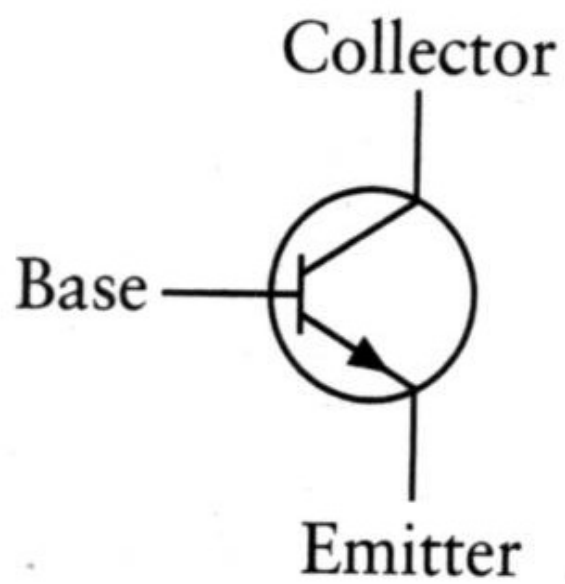
的晶状结构。这种半导体不是好的导体。但是半导体可以掺杂，意思是与某种杂质相混合。半导体很容易与其他杂质结合而变得

不纯。有一类杂质为原子的结合提供额外的电子，这种半导体叫 N 型半导体（N 表示

negative）；另一种类型的杂质掺杂生成 P 型半导体。

两个 N 型半导体中夹一个 P 型半导体可制成放大器，称作 NPN 晶体管，对应的三部分分别是集电极（Collector）、基极(Base)和发射极(Emitter)。

下面是一个 NPN 晶体管的示意图：



集电极

基极

发射极

基极上的一个小电压能控制一个很大的电压经过集电极到发射极。若基极上没有电压，它会有效截止晶体管。

晶体管通常被封装在一个直径大约为 1/4英寸的小金属容器里，有三个引脚伸出：



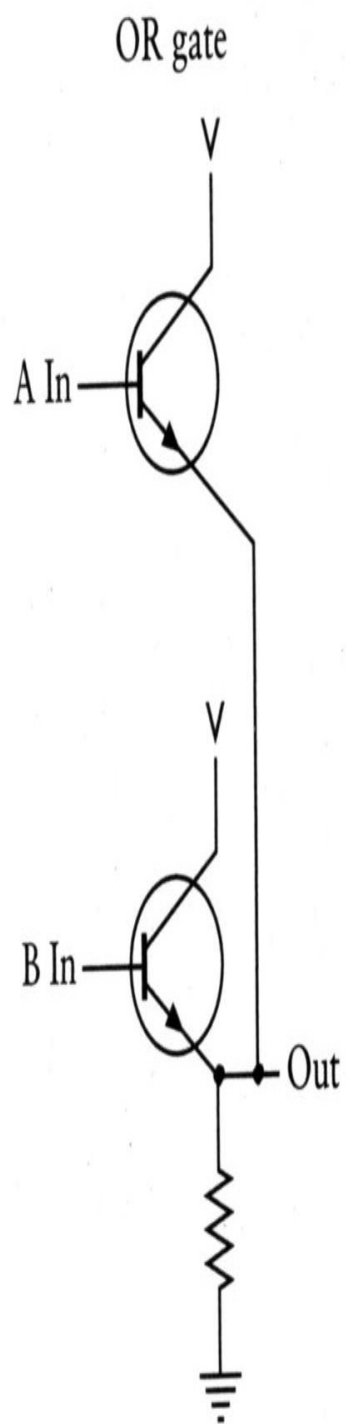
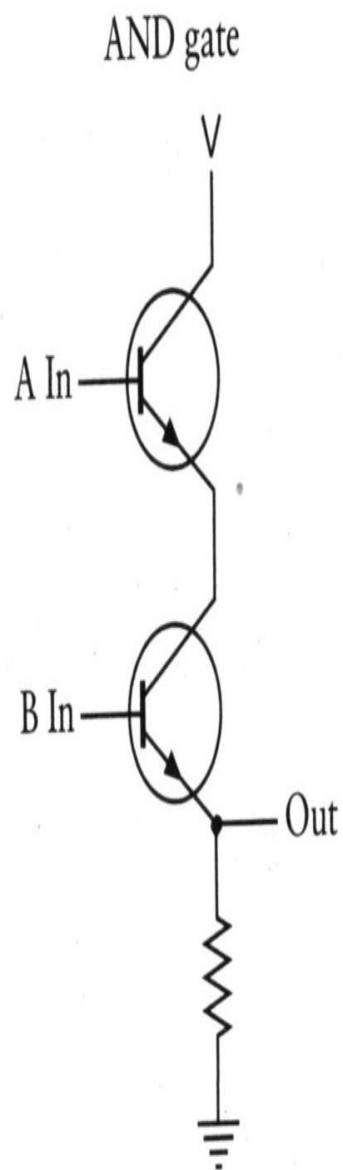
晶体管首创了固态电子器件，意思是晶体管不需要真空，可由固体做成，特别是指由半 导体和当今最普遍的硅制成。除了比真空管体积更小外，晶体管功耗小，发热少，并且更耐 用。携带一个电子管收音机很不方便。晶体管收音机靠小电池供电，并且不像电子管，它不 会变热。对于 1954年圣诞节早晨收到礼物的幸运者来说携带晶体管收音机已成为可能。那些 第一批袖珍晶体管收音机是由德克萨斯仪器公司制造的，该公司是半导体革命中的一个重要公司。

然而，晶体管的第一个商业应用是助听器。为了纪念贝尔一生为聋人的贡献， AT&T公 司允许助听器生产厂家不用付任何专利权税就可使用晶体管技术。第一台晶体管电视机出现 于1960年，而今天，电子管器件几乎已消失了。（然而，并未完全消失，一些高保真发烧友及 电子吉他手还是喜欢用电子管放大器来放大声音而不是用晶体管。）

1956年， Shockley离开贝尔实验室成立 Shockley半导体实验室。他迁到加利福尼亚的 Palo Alto—他的成长之地。他的公司是那里成立的第一家从事这种工作的公司。后来，其他半导 体及计算机公司也在那儿建立了业务，于是旧金山南部的这个地方也就成为现在非正式地称 作硅谷（ Silicon Valley）的地方。

真空管原是为放大信号而开发的，但它们也用来作为逻辑门的开关。晶体管也是如此。下面你将会看到一个由晶体管组成的与门在构造上很像用继电器组成的与门。只有当 A 输入 为1且B输入为1时，两个晶体管才都导通，输出才为 1。这里所示的电阻用来防止短路。

正如你所看到的，下图连接了两组晶体管，右边一个是或门，左边一个是与门。在与门 电路里，上面管子的发射极连接到下面管子的集电极。在或门电路里，两个晶体管的集电极 都连到电源，发射极连到一起：



或门

与门

A 输入

A 输入

B 输入

B 输入 输出

输出

因此，我们所学的有关由继电器构造逻辑门和其他部件的知识也适用于晶体管。继电器、电子管和晶体管最初主要用来放大信号，但也可用同样的方法连接成逻辑门来建造计算机。第一台晶体管计算机制造于 1956 年，短短几年内，新计算机的设计中就放弃使用电子管了。

这里有一个问题：晶体管当然使计算机更可靠，更小和更省电，但晶体管能使计算机的制造更简单吗？

并非如此。虽然晶体管使得在一个小空间里能安装更多的逻辑门，但你还得担心这些组件的互连。连接晶体管形成逻辑门像连接继电器和真空电子管一样困难，某种程度上，它甚至更困难，因为晶体管较小，不容易掌握。如果想用晶体管建造第 17 章所建造的计算机及 64KB 的 RAM 阵列，设计工作中的一部分将是设法发明某种能容纳所有组件的结构。这些劳动是乏味的，需要在数百万晶体管之间建立起数百万连接。

然而，就像我们所发现的，一些晶体管的连接会重复出现。许多对晶体管几乎都是先连接成门，门再连接成触发器、加法器、选择器或译码器，触发器再组成多位锁存器或 RAM 阵列。如果晶体管事先能连接成通用的结构，那么组装一台计算机就容易多了。

这种思想应该首先由英国物理学家 Geoffrey Dummer(1909 出生) 在 1952 年 5 月的一次演讲中提出，他说：

“我想预测一下未来。随着晶体的出现及半导体的日益广泛应用，现在似乎可以设想电子设备在一个固体块里而无需接线。这个块由隔离、传导、整理、放大四个层组成，电子功能通过各层的隔离区域直接连接起来。”

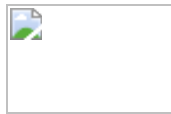
然而，真正可用的产品还不得不再等几年。

在对Dummer的预言毫不知情的情况下，1958年7月，德克萨斯仪器公司的 Jack Kilby（生于1923年）想到了在一个硅片上造出许多晶体管、电阻及其他电子组件的方法。6个月以后，即1959年1月，Robert Noyce（1927—1990）也找到了基本上相同的方法。Noyce原先曾为 Shockley 半导体实验室工作过，但在1957年，他和其他7位科学家离开实验室并成立了 Fairchild（仙童）半导体公司。

在技术史上，同时产生的发明很普遍，可能超出了你的想像。尽管 Kilby比Noyce早6个月发明了这项技术，并且德克萨斯仪器公司比仙童公司也早申请专利，但 Noyce还是首先获得了专利。法律上的争斗随之而来，但仅过了10年，问题就得到了令他们都满意的解决。尽管他们从未在一起工作过，但 Kilby和Noyce被认为是集成电路，或称 IC，更普遍的是叫芯片的共同发明者。

集成电路要经过复杂的工序才能生产出来，这些工序包括把很薄的硅晶片分层，精确地上涂料，和在不同的区域蚀刻形成微部件。尽管开发一种新的集成电路很昂贵，但收益来自于它的大量生产——生产得越多，就越便宜。

实际的硅晶片薄且很脆弱，为了保护芯片并提供某种方法使芯片中的部件与别的芯片连接，芯片必须安全地封装。集成电路的封装有几种不同的方式，但通常多采用矩形塑料双排直插式封装（或 dual inline package，DIP），有14、16或40个管脚从旁边伸出：



这是一个 16 管脚的芯片。如果你拿着芯片，并使芯片上的凹陷朝左时（如图），从芯片左 下角起环绕到右边至末端，管脚依次编号为 1~16，第 16 管脚在左上角。每一边管脚之间相距 刚好为 1/10 英寸。

20 世纪 60 年代，空间项目和军备竞赛刺激了早期的集成电路市场。在民用方面，第一个 包含有集成电路的商业产品是 1964 年由 Zenith 出售的助听器。1971 年，德克萨斯仪器公司开 始出售第一批袖珍计算器，同时 Pulsar 出售了第一块数字表（显然数字手表 中集成电路的封装 完全不同于刚才图示的例子）。随后出现了其 他很多种包含有集成电路的产品。

1965 年，戈登·E·摩尔（当时在仙童公司，后来是 Intel 公司的合伙 创始人）注意到技术 以这样一种方式在发展：1959 年后，可以集 成到一块芯片上的晶体管的数目每年都翻一番。他预测这种趋势 将继续下去。事实上后来这种趋势放慢了些，因此摩尔定律（最 终这样命名） 修改为预计每 18 个月在一个芯片上集成的晶体管数 量将增长一倍。这仍是一个令人惊异的增 长速度，这也就解释了 为什么家用电脑只用了短短几年好像就已过时了。一些人认为摩 尔定

律将继续适用到 2015 年。

早期，人们习惯于说小规模集成电路，即 SSI（small-scale integration），指那些少于 10 个 逻辑门的芯片；中规模集成电 路，即 MSI（10~100 个门）；大规模集成电路，即 LSI（100~ 5000 个门）。随后的术语为超大规模集成电路，即 VLSI（5 000 ~50 000 个门）；极大规模集 成电路，即 SLSI（50000~100 000 个门）；特大规模集成电路，即超过 100 000 个门。

本章的其余部分和下一章将把时钟停留在 20 世纪 70 年代中期，即 第一部《星球大战》发 行前的年代，那时，VLSI 刚刚出现，且 好几种技术用来制作构成集成电路的组件，每一种技 术有时被称

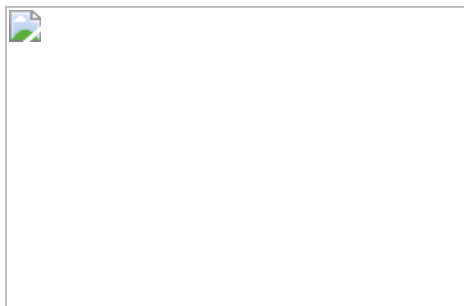
为一个 IC 家族。到 70 年代中期，两个“家族”流行开来： TTL 和 CMOS。

TTL 代表晶体管-晶体管逻辑。70 年代中期，如果你是一位数字电路设计工程师（即用 IC

来设计大的电路），一本 1.25 英寸厚的、德克萨斯仪器公司于 1973 年出版的名为《The TTL Data Book for Design Engineer》(TTL 工程师设计数据手册)的书(以下简称《TTL 数据手册》)就会是你书桌上的常客。这是一本关于德克萨斯仪器公司和其他公司出售的 TTL 集成电路 7400 系列的完整的参考书，之所以这样称呼是因为该家族的每个集成电路都以开头是 74 的数字标识。7400 系列中每一个集成电路由许多预先按特定方式连接的逻辑门组成。一些具备简单的预先连接的逻辑门的芯片可以用来建造更大的组件。还有一些芯片提供通用组件，如：触发

器、加法器、选择器和译码器。

7400 系列中第一个集成电路是号码 7400 本身，它在《TTL 数据手册》里描述为：“四个双输入正与非门”。这意味着这个特定的集成电路包含四个 2 输入与非门。它们之所以称为“正”与非门是因为有一个电压值与逻辑 1 对应，没有电压值与逻辑 0 对应。这个集成电路是一个 14 管脚的芯片，数据手册中的一个小图展示了管脚对应的输入和输出：



该图是芯片的俯视图（管脚在下面），小的凹陷在左边。

管脚14标为V

，等同于 V 符号，它表明是电压 (顺便说一下，大写字母 V 的双下标表明是

一个电源，下标中的 C 是指晶体管的集电极输入，它在内部连接到电源)。管脚 7 标为 **GND**，表示接地。在特定电路中所使用的任何集成电路都必须接电源和公共地。

对于TTL7400系列, V

CC

必须在 $4.75\sim 5.25\text{V}$ 之间，换句话说，电源电压必须在 $5\text{V}\pm 5\%$ 的

范围内。若电压低于 4.75V，芯片就无法工作；若超过 5.25V，芯片则会被烧坏。通常 TTL 器件不能使用电池供电，即使你刚好有一个 5V 的电池，电压也不可能刚好适合于芯片。TTL 通

常需要从墙上接入电源。

7400 芯片中每一个与非门有两个输入和一个输出，它们独立工作。前面曾把输入用 1（表

明有电压）和 0（表明无电压）加以区分，事实上与非门的输入电压可在 0（地）～ 5V（V

)

cc

之间变化。TTL 中，0～0.8V 之间的电压视为逻辑“0”，2～5V 之间的电压视为逻辑“1”，应当避免出现 0.8～2V 之间的输入电压。

TTL 的典型输出是 0.2V 代表逻辑“0”，3.4V 代表逻辑“1”。由于电压可能会有点儿变化，

集成电路的输入 / 输出有时称作“高”或“低”，而不是“1”或“0”。此外，低电平也可以表示逻辑“1”，高电平也可以表示逻辑“0”，这种说法称为负逻辑。当 7400 芯片称作“四个双输入正与非门”时，“正”就表示上面讲到的正逻辑。

如果 TTL 的典型输出是 0.2V 代表逻辑“0”，3.4V 代表逻辑“1”，则输出确实是在输入范围内，即逻辑“0”在 0～0.8V、逻辑“1”在 2～5V 之间，这也是 TTL 能隔离噪声的原因。一个“1”输出即使下降 1.4V 后，电压仍可以高到作为“1”电平输入；一个“0”输出升高 0.6V 后，电压仍可以低到作为“0”电平输入。

了解一个集成电路最重要的事实可能是它的延迟时间，这指的是输入变化反应到输出所花费的时间。

芯片的延迟通常以纳秒来计算，纳秒缩写为 ns。1纳秒非常短暂，千分之一秒为 1毫秒，

百万分之一秒为 1 微秒，十亿分之一秒才是 1纳秒。7400芯片与非门的延迟时间要保证少于

22纳秒，也就是 0.000000022秒，或十亿分之 22秒。不能感觉纳秒的长短不仅仅是你一人，地球上任何人除了对纳秒有智力上的理解之外就

没有什么了。纳秒比人所经历的任何事物要短暂得多，所以它们永远也不会被理解，任何解释都只会使纳秒变得更难以捉摸。例如，你拿着一本书离你的脸有一英尺远，那么 1纳秒就是

光从书到你的眼所用的时间。现在，你能说你对纳秒有了更好的认识？然而，纳秒在计算机中出现是可能的。正如在第 17章所见，计算机处理器很笨拙地做着

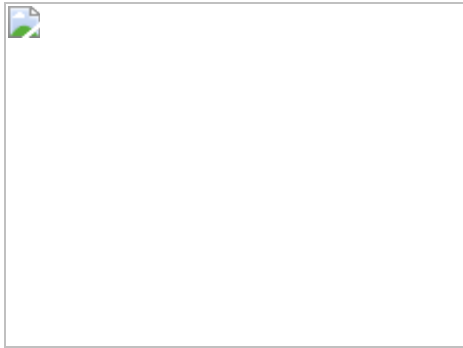
简单的事情——从存储器中取一个字节到寄存器，把一个字节同另一个字节相加，再把结果

存回存储器。计算机（不是第 17章中的计算机，而是现在使用的计算机）能做大量事情的唯一原因就是那些操作能迅速执行。引用 Robert Noyce 的话：“当你理解了纳秒之后，在概念上 计算机操作就相当简单了。”

我们继续细读《TTL工程师设计数据手册》，就会在书中看到许多熟悉的小条目。7402芯片有四个双输入或非门，7404有六个反相器，7408有四个双输入与门，7432有四个双输入或门，7430有一个 8输入与非门：

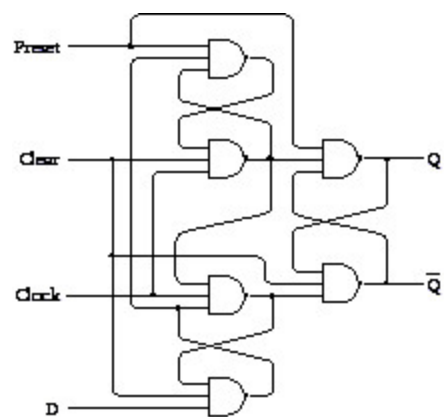


图中缩写 NC表示该引脚没有连接到内部电路。



7474芯片是听起来很熟悉的一个芯片，它是“带预置和清零的双 D 型正边沿触发器”。如下图所示：

TTL数据手册甚至还包含这个芯片的每个触发器的逻辑图：



预置

清零

时钟

你会发现这与第 14 章结尾处的图很相似，只是第 14 章的图中使用的是或非门。《TTL 数据手册》中的逻辑表也有一点点不同：



输入 输出

上表中，“H”代表高电平，“L”代表低电平。你也可以把它们想成是 1或0。在上述触发器里，预置与清零输入通常为“0”，在这里它们为“1”。

继续翻阅《TTL数据手册》，你会发现 7483芯片是一个 4位二进制全加器，74151是一个8-1

数据选择器，74154是4-16译码器，74161是同步 4位二进制计数器，74175是四个带清除功能的D型触发器。你可以选择这些芯片中的两个做一个 8位锁存器。

所以现在你该明白从第 11章起使用的各种各样的组件是如何来的了，它们都是从《TTL

工程师设计数据手册》中得来的。作为一名数字电路设计工程师，需要花费大量的时间去通读《TTL数据手册》，了解要使

用的TTL芯片的类型。一旦掌握了你所需要的工具，你就可以用 TTL芯片实际组装第 17章所示例的计算机。把芯片连接起来要比连接晶体管容易得多，然而你可能不想用 TTL组成64KB的

RAM阵列。在 1973年的《TTL数据手册》中，所列最大容量的 RAM芯片才 256×1位，需要

2048个这种芯片才能组成 64KB的RAM！TTL远不是组织存储器的最好技术，第 21章将要更多地谈到关于存储器的事情。

你可能也想用好一些的振荡器。可以将 TTL反相器的输出端连到输入端，但使用一个事先可预测频率的振荡器要更好一些。构造一个这样的振荡器很容易，就是使用一个石英晶体，

在一块小片上引出两条线。这些晶体在特定的频率下产生振荡，通常每秒至少 100万周。每秒一百万周称为兆赫，缩写为 MHz。如果第 17章所示的计算机是用 TTL构造的话，那么它在

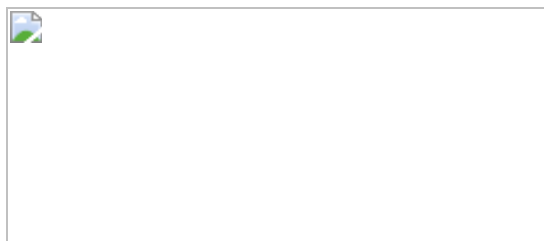
10MHz的时钟频率下可能会运行地很好。每一条指令需要 400纳秒的执行时间。当然，这已经比用继电器工作时所能想像的要快多了。

另一个流行的芯片家族是（现在仍然是）CMOS，它代表由金属氧化物填充的半导体。

如果你是 70 年代中期用 CMOS集成电路进行电路设计的业余爱好者，你可能会使用一本由 National Semiconductor(国家半导体公司)出版的参考书，它在你所在地方的电器行就能见到，书名为《CMOS Databook》。此书包含了 CMOS集成电路 4000系列的信息。

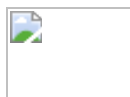
TTL的电源要求是在 4.75~5.25伏之间，但 CMOS则可以是 3~18伏之间的任何数值，范围多大呀！此外，CMOS比TTL功耗要小，这就可以使用电池来驱动小型 CMOS电路。CMOS的缺陷是速度慢。例如，CMOS 4008 4位全加器在 5伏电压下只能保证 750纳秒的延迟。当电源电压升高时，速度加快——10伏时，延迟为 250纳秒；15伏时，延迟为 190纳秒。但是 CMOS设备不能接近于 TTL 4位加法器，TTL 4位加法器的延迟为 24纳秒。（25年前，在 TTL的速度和CMOS的低功耗之间的权衡是很清楚的，今天，也有低功耗的 TTL和高速率的 CMOS。）

实践上，你可以在塑料面包板上连接这些芯片进行实验：



每一个有 5 个孔的短行在塑料板下是电导通的。你把芯片插在面包板上，并使芯片跨在中间的长槽上，管脚插入槽两边的孔中。集成电路的每一个管脚都与其他 4 个孔电连接。你可以将线插入其余孔中来连接芯片。

你也可以使用一种称为线缠绕的技术使芯片的连接更加牢固。每一个芯片插入带有长方形柱子的插座上：



每一个柱子对应于芯片的一个管脚，插座本身也插入打孔的板上。在板的另一边，你会看到特殊的用线缠绕的插槽紧紧包裹着围绕柱子的绝缘线。柱子的方形边缘穿破绝缘层并使它和导线电连接。

如果你实际在使用集成电路来构造一个电路，就要使用一块印刷线路板。以前，这是业余爱好者可以做的事情。板子上有孔，并覆盖一层薄的铜箔。首先，你要在需要保护的区域的铜箔上涂上防酸物，并用酸腐蚀其余部分，然后你把集成电路插座（或集成电路本身）直接焊在板的铜上。由于集成电路之间有许多内部连

接，一层铜箔通常是不够用的，商业制造的印刷线路板有许多内部互连的层。

到70年代早期，已可以在一块电路板上用集成电路构造一个完整的计算机处理器。把整个处理器做在一块芯片上已只是时间问题。尽管德克萨斯仪器公司 1971年为单片计算机申请了专利，但实际的制造荣誉却属于 Intel——一家于 1968年，由前仙童雇员 Robert Noyce 和 Gordon Moore 建立的公司。Intel的第一个主要产品是 1970年生产的可存储 1024位的存储器芯片，在当时这是可做在一块芯片上的最大存储容量。

Intel在为由日本的 Busicom公司生产的可编程计算器设计芯片时，决定采用不同的方法。正如Intel公司的工程师 Ted Hoff写的：“不是想使他们的设备成为一个带有编程能力的计算器，

而是想使它作为通常目的计算机可编程为一个计算器”。这就产生了 Intel 4004，第一个“芯片上的计算机”或微处理器。1971年11月，4004投入使用，它带有 2300个晶体管。（根据摩尔定

律，18年后微处理器应该有 4000倍数量的晶体管，即大约 1000万个，这是相当准确的预计。）有了晶体管的数量，下面将描述 4004的其他三个很重要的特性。自 4004诞生以来，这三

个指标经常用来作为微处理器相互比较的标准。

第一，4004是4位的微处理器。这意味着处理器的数据通路宽度只有 4位，做加、减法运算时，它一次只处理 4位。相比较，第 17章中开发的计算机有 8位数据通路，所以它是 8位处理

器。正如我们将看到的 4 位微处理器很快就被 8 位微处理器所超越，没有人会停滞不前。70 年代后期，16 位微处理器出现了。回想一样第 17 章的内容，以及在 8 位处理器上进行两个 16 位数相加所需要的指令代码，你就会欣赏 16 位处理器带给你的好处。80 年代中期，32 位微处理器出现了，并从那以后一直作为家用计算机的主流微处理器。

第二，4004 有最大每秒 108 000 周的时钟频率，即 108KHz。时钟频率是可连接到微处理器并能运行的振荡器的最大频率。再要快的话，微处理器就可能出错。到 1999 年，家用计算机微处理器的时钟频率已达到 500MHz——大约 4004 运行速率的 5000 倍。

第三，4004 可寻址的存储器是 640 个字节。这看起来是一个小得可笑的数字，然而这和当时可用的存储芯片的容量是相匹配的。下一章你就会看到，两年内微处理器可达到 64KB 的寻址空间，这是第 17 章所提及的机器的容量。到 1999 年，Intel 的微处理器已达 64TB 的可寻址空间，尽管大多数人家用电脑的 RAM 容量还低于 256MB。

这三个数字不会影响一台计算机的能力。例如，一个 4 位处理器要进行 32 位数的加法，只要简单地按 4 位一次进行。某种意义上，所有数字计算机都是相同的。如果一个处理器的硬件能做别的处理器做不了的，那么别的处理器可以用软件实现，最终它们能做同样的事情。这也就是图灵 1973 年的论文里有关计算能力的含义。

然而，处理器根本的不同是在速度上。同时，速度也是我们为什么使用计算机的一个重要原因。

最大时钟频率是影响处理器总体速度的一个显著因素，时钟频率决定了每一条指令的执行速度。处理器的数据宽度也影响其执行速度。虽然一个 4 位处理器可进行 32 位数的加法运算，但它的执行速度不可能与 32 位处理器一样。然而，令人迷惑的是，处理器

可寻址的最大存储器容量也是影响速度的一个因素。最初，寻址空间看起来好像与处理器速度无关，而只反映了处理器在执行某些需要大量存储空间的功能时处理器的能力限度。但处理器通过利用存储器地址来控制用于保存或提取信息的其他媒体，可避开存储容量的限制。（例如，假设写到某个存储地址的每个字节实际上都是在纸带上穿孔，从存储地址读的每个字节都是从纸带上读。）然而这种做法减慢了整个计算机的处理速度——又是速度问题。

当然，这三个数字都只是粗略地显示了微处理器的运行速度。这些数字没有告诉任何有关微处理器内部体系结构或机器码指令的效率和能力的问题。处理器越来越复杂，许多以前用软件来实现的普通工作现在可以用微处理器来实现。我们在后面的各章中可看到这种趋势的一些例子。

即使所有的数字计算机都具有同等的能力，即使它们只能做与图灵设计的原始计算机一样的工作，处理器的执行速度最终也会影响计算机系统的总体用途。例如，比人类大脑的计算速度还慢的计算机是毫无用处的。当我们在现代计算机的屏幕上看电影时，如果处理器需要花费1分钟的时间来处理每一帧，我们也是无法忍受的。

回到20世纪70年代中期，先不说4004的局限性，但毕竟它是一个开始。1972年4月，Intel发布了8008——一个8位微处理器，时钟频率为200KHz，可寻址16KB的存储空间。（仅用三个数来总结一个处理器是多么容易。）后来，1974年5月期间，Intel和Motorola公司同时发布了对8008进行改进的微处理器，这两种芯片改变了整个世界。

第 19 章 两种典型的微处理器

微处理器——集成计算机中央处理器（CPU）的所有组件在一个硅芯片上——诞生于 1971 年。它的产生有一个很好的开端：第一个微处理器是 Intel 4004，其中有 2300 个晶体管。今天，差不多 30 年过去了，为家用计算机所制造的微处理器中将近有 10 000 000 个晶体管。

微处理器实际的作用基本上保持不变。现在的芯片上附加的上百万个晶体管可以做许多有趣的事情，但在微处理器最初的探索过程中，这些事情更多的是分散我们的注意力而不是给我们以启迪。为了对微处理器的工作情况获得更清晰的认识，让我们先看一下最初的微处理器。这些微处理器出现在 1974 年。在该年度，Intel 公司在 4 月推出了 8080，Motorola（摩托罗拉）——从 20 世纪 50 年代开始生产半导体和晶体管产品的公司——在 8 月份推出了 6800。它们并非是那年仅有的微处理器。同样是在 1974 年，德克萨斯仪器公司推出了 4 位的 TMS 1000，用在许多计算器、玩具和设备上；National Semiconductor（国家半导体公司）推出了 PACE，它是第一个 16 位的微处理器。然而，回想起来，8080 和 6800 是两个最具有重大历史意义的芯片。Intel 设定 8080 最初价格为 \$ 360，这是对 IBM System/360 的一个讽刺。IBM System/360 是一个大型机系统，由许多大公司使用，要花费几百万美元。（今天，你只花 \$ 1.95 就可以买到一个 8080 芯片。）这并不是说 8080 可以与 System/360 相提并论，但不用几年，IBM 公司也将会

正视这些很小的计算机。

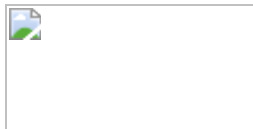
8080是一个 8位的微处理器，有 6000个晶体管，时钟频率为 2MHz，可寻址 64KB的存储空间。6800（今天也只卖 \$ 1.95）有大约 4000个晶体管，也可寻址 64KB的存储空间。第 1代6800 的时钟频率为 1 MHz，但到 1977年Motorola公司发布新款的 6800 时，其时钟频率已为 1.5MHz 和2 MHz。

这些芯片称作“单芯片微处理器”，不太准确的名称为“一个芯片上的计算机”。处理器只是整个计算机的一部分。此外，计算机至少还需要一些随机访问存储器（RAM）、供人们输入信息到计算机的方法（输入设备），供人们从计算机获取信息的方法（输出设备），以及其他可把所有这些东西连接在一起的芯片。这些组件将在第 21章详细介绍。

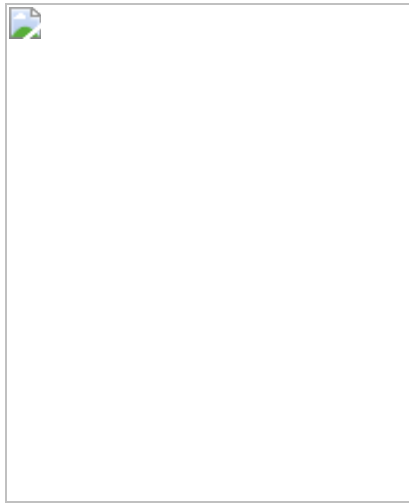
从现在起，我们只考察微处理器。描述微处理器时，通常是用框图来画微处理器的内部组件及它们是如何连接的。但在第 17章已有够多的图了，现在，我们将通过观察处理器与外界的相互作用来了解它的内部。换句话说，为了弄清楚它的作用，可以把微处理器看成是一个黑盒子，它的内部操作不需要做详细研究。我们可以通过测试芯片的输入和输出信号，特别是芯片的指令集来掌握微处理器的功能。

8080和6800都是40管脚的集成电路。这些芯片最普通的 IC封装大约是 2英寸长，半英寸宽，

1/8英寸厚：



当然，你看到的只是外包装。位于其内部的硅晶片非常小，就拿早期的 8 位微处理器来说，其硅晶片小于 1/4 平方英寸。外包装保护硅晶片并通过管脚提供对芯片的输入和输出点的访问。下图显示了 8080 的 40 个管脚的功能：



本书的所有电气或电子设备都需要某种电源供电。8080 的一个特别之处在于它需要三种电源电压：管脚 20 必须连到 5 伏电源上，管脚 11 连到 -5 伏电源上，管脚 28 连到 12 伏电源上；管脚 2 接地（1976 年，Intel 发布了 8085 芯片，简化了这些电源需求）。

其余管脚都画有箭头。从芯片中出来的箭头表示输出信号，这是由微处理器控制的信号，计算机中其余芯片对此作出响应。指向芯片的箭头表示输入信号，这是来自于其他芯片的信号，8080 对它们做出响应。有些管脚既是输入又是输出。

第 17 章的处理器需要振荡器使它工作。8080 需要两个不同的 2 MHz 同步时钟输入，在 22

和 15 管脚上分别标记为 \overline{CLK} 和 $\overline{CLK2}$ 。这些信号可以很方便地由 Intel 公司生产的 8224 时钟信号发生

器提供。给这个芯片连上一个 18 MHz 的石英晶体，剩下的工作它基本上可以完成。

一个微处理器通常有多个输出信号来寻址存储空间，这种信号的数目与微处理器可寻址的存储器空间的大小直接相关。8080 有 16 个地址信号，标为 A0~A15，具有寻址 2_{16} 即 65 536 字节的存储空间的能力。

8080 是一个 8 位微处理器，一次可从存储器中读出、写入 8 位数据。它包括 8 个数据信号

D₀~D₇，这些信号是在此芯片中仅有的几个既作为输入又作为输出的信号。当微处理器从存储

0 7

器读数据时，这些管脚作为输入；当微处理器向存储器写数据时，这些管脚作为输出。

微处理器的另外 10 个管脚是控制信号。例如，RESET 输入用来复位微处理器。输出信号

— -

WR 表示微处理器要向 RAM 中写数据。（WR 信号对应于 RAM 阵列的写入输入。）另外，当芯

片读取指令时，其他控制信号会在某个时候出现在 D₀~D₇ 管脚。由 8080 构成的计算机系统通常

0 7

使用 8228 系统控制芯片来锁存这些附加的控制信号。后面将会讲述一些控制信号。由于 8080

的控制信号非常复杂，因此，除非你想基于 8080 芯片来实际设计计算机，否则最好不要用这些控制信号来折磨自己。

假定8080微处理器连接了 64KB的存储器，这样可以不通过微处理器来读写数据。8080芯片复位后，它从存储器的地址 0000h处读取该字节，送到微处理器中。这可以通过

在地址信号端 $A \sim A$ 程叫作取指令。

0

15

输出 16个0来实现。它读取的字节必须是 8080指令，这种读取字节的过

在第17章构造的计算机里，所有指令（除了停止指令 **HLT**）都是3个字节长，包括一个操作码和两个字节的地址。在 **8080** 中，指令长度可以是1个字节、2个字节或3个字节。有些指令可使 **8080** 从存储器的某一位置处读出一个字节送到微处理器中；有些指令可使 **8080** 从微处理器中把数据写入存储器的某一位置处；其他指令可使 **8080** 不使用 **RAM** 而在内部执行。第一条指令执行完后，**8080** 访问存储器中的第二条指令，依此类推。这些指令组合在一起构成一个计算机程序，用来完成一些自己感兴趣的事情。

当 **8080** 运行在最高速度即 **2 MHz** 时，每个时钟周期为 500 纳秒（1 除以 2 000 000 周等于 0.000000500 秒）。第17章中的每条指令都需要 4 个时钟周期，**8080** 的每条指令则需要 4~18 个时钟周期，这意味着每条指令的执行时间为 2~9 微秒（即百万分之一秒）。

了解微处理器功能的最好方法可能是在系统方式下测试其完整的指令集。第17章最后出现的计算机仅有 12 条指令。一个 8 位微处理器很容易就有 256 条指令，每个操

作码对应于某个 8 位值。（如果一些指令有 2 个字节的操作码，则实际会有更多的指令）。**8080** 虽没有那么多，但它也有 244 条操作码。这看起来似乎很多，但总的来说，却又不比第 17 章中的计算机功能多多少。例如，如果想用 **8080** 做乘法或除法，仍然需要写一段小程序来实现。

第17章中讲过，处理器指令集的每个操作码都和某个助记符相联系，有些助记符之后可能还有操作数。但这些助记符仅用来方便地表示操作码。处理器只读取字节，它并不懂组成这些助记符的字符的含义。

第17章中的计算机有两条很重要的指令，称作装载（**Load**）和保存（**Store**）指令。这些指令都占用三个字节的存储空间。装载指令的第一个字节是操作码，操作码后的两个字节表示 16 位地址。处理器把在此地址中的字节送到累加器。同样，保存指令把累加器中的内容存储到指令指定的地址处。

下面，我们用助记符来简写这两个操作：

LOD A, [aaaa]

STO [aaaa], A

在此，**A**表示累加器（装载指令的目的操作数，保存指令的源操作数），**aaaa**表示一个 16

位的存储器地址，通常用 4位十六进制数来表示。

8080中的8位累加器称作 **A**，就像第 17章中的累加器。正如第 17章中的计算机一样，8080 也有两条与装载和保存指令功能一样的指令。8080中这两条指令的操作码为 32h和3Ah，每个 操作码后有一个 16位地址。8080的助记符为 **STA**（代表存储累加器的内容）和 **LDA**（代表装 载到累加器）：

操作码 指令

32 STA [aaaa],A 3A LDA A,[aaaa]

除了累加器，8080微处理器内部还包括 6个寄存器（**register**），每个寄存器可以保存 8位的

值。这些寄存器和累加器非常相似，事实上，累加器被看作是一种特殊的寄存器。和累加器

一样，这 6 个寄存器也是锁存器。处理器可以把数据从存储器传送到寄存器，也可以把数据从寄存器送回到存储器。然而，这些寄存器没有累加器的功能强大。例如，当两数相加时，其结果通常送往累加器而非其中一个寄存器。

在8080中，除累加器外的 6 个寄存器的名字分别为 B，C，D，E，H和 L。人们通常问的第一个问题是“用 F和G干什么？”，第二个问题是“用 I，J和K又要做什么？”，答案是使用寄存器H和L具有某种特殊的含义。H代表高（High），L代表低(Low)。通常把 H和L的8位合起来记作HL来表示一个 16位寄存器对，H作为高位字节，L作为低位字节。这个 16 位值通常用来寻址存储器。后面我们将看到它的简单用法。

所有这些寄存器都是必需的吗？为什么不在第 17章中的计算机中用到它们呢？从理论上说，它们并非必需，但是使用它们会很方便。许多计算机程序在同一时刻要用到几个数据，如果所有这些数据都存储在微处理器的寄存器中而非存储器中，执行程序将会更快，因为程序访问存储器的次数越少，那么它的运行速度也就越快。

8088指令中，有一个至少 63个指令码供一条 8080指令使用的指令，它就是 MOV指令，即 Move的简写。该条指令只有一个字节，用于把一个寄存器中的内容传送到另一个寄存器中

（或同一个寄存器中）。使用大量 MOV指令是设计带有 7个寄存器（包括累加器）的微处理器的正常结果。

下面是前 32条MOV指令。记住目的操作数在左边，源操作数在右边：

操作码	指令	操作码	指令
40	MOV B , B	50	MOV D , B

41	MOV B , C	51	MOV D , C
42	MOV B , D	52	MOV D , D
43	MOV B , E	53	MOV D , E
44	MOV B , H	54	MOV D , H
45	MOV B , L	55	MOV D , L
46	MOV B , [HL]	56	MOV D , [HL]
47	MOV B , A	57	MOV D , A
48	MOV C , B	58	MOV E , B
49	MOV C , C	59	MOV E , C
4A	MOV C , D	5A	MOV E , D
4B		5B	
4C	MOV	5C	MOV
4D	C , E MOV	5D	E , E MOV
4E	C , H MOV	5E	E , H MOV

	C , L MOV		E , L MOV
	C , [HL]		E , [HL]
4F	MOV	5F	MOV
	C , A		E , A

这些都是很方便的指令。当一个寄存器中有值时，可以把它传送到其他寄存器中。注意，上述指令中有四条指令用到 **HL** 寄存器对，如：

MOV B , [HL]

前面列出的 **LDA** 指令把一个字节从存储器中传送到累加器中，这个字节的 **16** 位地址直接跟在 **LDA** 操作码的后面。这里的 **MOV** 指令把一个字节从存储器中传送到寄存器 **B** 中，但被装载到寄存器中的字节的地址是保存在寄存器对 **HL** 中。**HL** 寄存器是怎样得到 **16** 位存储器地址的呢？它可以通过多种方法来实现，或许是通过某种方法计算出来的。

总之，这两条指令

```
LDA      A ,  [aaaa]
```

```
MOV      B ,  [HL]
```

都把一个字节从存储器中装载到微处理器中，但它们用两种不同的方法来寻址存储器地址。第一种方法叫作直接寻址方式，第二种方法叫作间接寻址方式。

第二批32条MOV指令表明用 HL寻址的存储器地址也可以作为目的操作数：

操作码	指令	操作码	指令
60	MOV H , B	70	MOV [HL] , B
61	MOV H , C	71	MOV [HL] , C
62	MOV H , D	72	MOV [HL] , D
63	MOV H , E	73	MOV [HL] , E
64	MOV H , H	74	MOV [HL] , H
65	MOV H , L	75	MOV [HL] , L

66	MOV H , [HL]	76	HLT
67	MOV H , A	77	MOV [HL] , A
68	MOV L , B	78	MOV A , B
69	MOV L , C	79	MOV A , C
6A	MOV L , D	7A	MOV A , D
6B	MOV L , E	7B	MOV A , E
6C	MOV L , H	7C	MOV A , H
6D	MOV L , L	7D	MOV A , L
6E	MOV L , [HL]	7E	MOV A , [HL]
6F	MOV L , A	7F	MOV A , A

其中一些指令如：

MOV A, A

做的是无用的事，而像：

MOV [HL], [HL]

这样的指令是不存在的。和这条指令相对应的操作码实际上是停止指令。观察这些 MOV 操作码更明显的方法是考察它的位模式，MOV 操作码由 8 位组成：

01dddsss

其中字母ddd 代表指代目的操作数的3位代码，sss代表指代源操作数的3位代码。这3位代码是：

000= 寄存器 B

001= 寄存器 C

010= 寄存器 D

011= 寄存器 E

100= 寄存器 H

101= 寄存器 L

110= HL 中保存的存储器地址中的内容

111= 累加器A

例如，指令：

MOV L, E

相应的操作码表示为 01101011，或6Bh。可以通过检查前面的表来验证。

因此可能在 8080内部某个地方，标有 sss的3位标识用在 8-1数据选择器中，标有 ddd的3位 标识用于控制 3-8译码器，此译码器用来决定哪一个寄存器锁存了一个值。

也可能使用寄存器 B和C来构成一个 16位寄存器对 BC，用寄存器 D和E来构成一个 16位寄 存器对 DE。如果每一个寄存器对都包含用于装载或保存一个字节的存储器地址，则可以使用 下述指令：

操作码	指令	操作码	指令
02	STAX [BC] ， A	0A	LDAX A ， [BC]
12	STAX [DE] ， A	1A	LDAX A ， [DE]

另一种类型的传送指令叫做传送立即数，指定的助记符为 MVI。传送立即数指令占两个 字节，第一个是操作码，第二个是 1个字节的数据。此字节从存储器中传送到一个寄存器中或 由HL寻址的存储单元中。

操作码	指令
06 MVI	B ， xx
0E	MVI C ， xx
16 MVI	D ， xx

1E MVI E , xx

26 MVI H , xx

2E MVI L , xx

36 MVI[HL] , xx

3E MVI A , xx

例如，当指令：

MVI E, 37h

执行后，寄存器 E 中包含有字节 37h。这就是第三种寻址方式，即立即数寻址方式。32 个操作码的集合完成四种基本算术运算，那是在第 17 章开发处理器时我们就已熟悉的

运算，即加法（ADD）、进位加法（ADC）、减法（SUB）和借位减法（SBB）。所有情况中，累加器是两个操作数之一，也是结果的目的地址。

操作码	指令	操作码	指令
80	ADD A , B	90	SUB A , B
81	ADD A , C	91	SUB A , C
82	ADD A , D	92	SUB A , D
83	ADD A , E	93	SUB A , E
84	ADD A , H	94	SUB A , H
85	ADD A , L	95	SUB A , L
86	ADD A , [HL]	96	SUB A , [HL]
87	ADD A , A	97	SUB A , A

88	ADC A , B	98	SBB A , B
89	ADC A	99	SBB A
8A	, C	9A	, C
	ADC A , D		SBB A , D
8B	ADC A	9B	SBB A
8C	, E	9C	, E
8D	ADC A	9D	SBB A
8E	, H ADC A	9E	, H SBB A
8F	, L ADC A [HL] ADC A A	9F	, L SBB A [HL] SBB A A

假设A中是35h,寄存器B中是22h,当指令:

```
SUB     A, B
```

执行后,累加器中的结果为 13h。

若A中的值为 35h,寄存器 H中的值为 10h, L中的值为 7Ch, 存储器地址 107Ch中的值为 4A h, 则指令:

```
ADD A, [HL]
```

把累加器中的内容 (35h) 和通过寄存器对 HL寻址得到的值 (4Ah) 相加, 并把结果

(7Fh) 保存到累加器中。

ADC和SBB指令允许 8080加/减16位、24位、32位和更多位的数。例如, 假设寄存器对 BC

和DE都包含16位数, 你想把它们相加, 并把结果存到 BC中。下面是具体做法:

```
MOV     A, C           ; 低位字节
```

```
ADD     A, E
```

```
MOV     C, A
```

```
MOV     A, B           ; 高位字节
```

```
ADC     A, D
```

MOV

B, A

其中有两条加法指令，**ADD**指令用于低位字节相加，**ADC**指令用于高位字节相加。第一条加法指令的进位位包含在第二条加法指令中。因为只能利用累加器进行加法运算，所以在这么短的代码中也需要至少4条**MOV**指令。许多**MOV**指令常常出现在8080代码中。

该是谈论8080标志位的时候了。在第17章的处理器中，已有进位标志位**CF**和零标志位**ZF**。8080还有3个标志位，即符号标志位**SF**、奇偶标志位**PF**和辅助进位标志位**AF**。所有标志位都保存在另一个叫作程序状态字（**PSW: program status word**）的8位寄存器中。像**LDA**、**STA**和**MOV**这样的指令不影响标志位，而**ADD**、**SUB**、**ADC**和**SBB**指令却要影响标志位，影响的方式如下：

- 当运算结果最高位为1时，符号标志位**SF**为1，表示结果为负。
- 当结果为0时，零标志位**ZF**为1。
- 当运算结果中“1”的个数为偶数时，奇偶标志位**PF**=1；当运算结果中“1”的个数为奇数时，奇偶标志位**PF**=0。**PF**有时用来粗略地检测错误，此标志位在8080程序中不常用。
- 当**ADD**或**ADC**运算产生进位或**SUB**与**SBB**运算不发生借位时，进位标志位**CF**=1。（这点不同于第17章中的计算机进位标志的实现。）
- 当操作结果的低4位向高4位有进位时，辅助进位标志位**AF**=1。此标志位只用在**DAA**

（十进制调整累加器）指令中。有两条指令直接影响进位标志位**CF**：

操作
码

指令

含
义

37

STC

置
CF
为
1

3F

CMC

CF
取
反

第17章中的计算机可执行 **ADD**、**ADC**、**SUB**和**SBB**指令（尽管没什么灵活性），但8080还

可以进行逻辑运算 **AND**（与）、**OR**（或）和 **XOR**（异或）。算术运算和逻辑运算都可通过处理器的算术逻辑单元（**ALU**）来执行：

操作码

指令

操作码

指令

A0

AND A , B

B0

OR A , B

A1

AND A
, C

B1

OR A
, C

A2

AND A
, D

B2

OR A
, D

A3

AND A
, E

B3

OR A
, E

A4

AND A
, H

B4

OR A
, H

A5

AND A
, L

B5

OR A
, L

A6

AND A
, [HL]

B6

OR A
, [HL]

A7

AND A
, A

B7

OR A
, A

A8

XOR A ,
B

B8

CMP
A ,
B

A9

XOR A ,
C

B9

CMP
A ,
C

AA

XOR A ,
D

BA

CMP
A ,
D

AB	XOR A , E	BB	CMP A , E
AC	XOR A , H	BC	CMP A , H
AD	XOR A , L	BD	CMP A , L
AE	XOR A , [HL]	BE	CMP A , [HL]
AF	XOR A , A	BF	CMP A , A

AND、**XOR**和**OR**指令按位运算，即逻辑操作只是单独地在对应位之间进行。例如：

```
MVI A, 0Fh MVI B, 55h AND A, B
```

累加器中的结果将为 **05h**。如果第三条指令为 **OR**运算，则结果为 **5Fh**；如果第三条指令为 **XOR**运算，则结果为 **5Ah**。

CMP（比较）指令与 **SUB** 指令基本上一样，除了结果不保存在累加器中。换句话说，**CMP**执行减法操作再把结果扔掉。这是为什么？是因为标志位。根据标志位的状态可知道所比较的两数之间的关系。例如，当如下指令：

```
MVI B, 25h CMP A, B
```

执行完时，**A**中的内容没有改变。然而，若 **A**中的值为 **25h**，则 **ZF**标志置位；若 **A**中的值 小于**25h**，则**CF = 1**。

这8个算术逻辑运算也可以对立即数进行操作：

操作
码

指
令

操作
码

指
令

C6
CE
D6
DE

ADI
A
,
xx
ACI
A
,
xx
SUI
A
,
xx
SBI
A
,
xx

E6
EE
F6
FE

ANI
A
,
xx
XRI
A
,
xx
ORI
A
,
xx
CPI
A
,
xx

例如，上面列出的两条指令也可以用下面的指令来替换：

CPI A, 25h

下面是其他两条 8080指令：

操作码 指令

27 DAA

2F CMA

CMA即complement accumulator，它对累加器中的值进行取反操作。每个 0变为1，每个 1

变为 0。如果累加器中的值为 01100101，CMA指令使它变为 10011010。也可以用下述指令来使累加器按位取反：

```
XRI    A, FFh,
```

DAA即Decimal Adjust Accumulator，如前所述，它可能是 8080中最复杂的一条指令。微处理器中有一个完整的小部件专门用于执行这条指令。

DAA指令帮助程序员用BCD码表示的数来进行十进制算术运算。在BCD码中，每一小块数据的范围在 0000~1001之间，对应于十进制的 0~9。利用BCD码格式，每 8位字节可存储两个十进制数字。

假设累加器中的值为BCD码的27h。由于是BCD码，则它实际上指的是十进制的 27。（十六进制的 27h等于十进制的 39。）再假定寄存器 B 中的值为BCD码的94h。如果执行指令：

```
MOV    A, 27 h
```

```
MOV    B, 94 h
```

```
ADD    A, B
```

累加器中的值将为 BB h，当然，它不是BCD码，因为BCD码中的每一块不能超过 9。但是，现在执行指令：

```
DAA
```

则累加器中的值为 21h，且CF = 1，这是因为 27和94的十进制和为 121。如果想进行BCD

码的算术运算，这样做是相当方便的。

经常需要对一个数进行加 1或减 1操作。在第 17章的乘法程序中，我们需要对一个数减 1，使用的方法是加上 FFh，它是 -1的2的补码。8080中包含特殊的用于寄存器或存储单元的加 1 指令（称作增量）和减 1指令（称作减量）：

操作码	指令	操作码	指令
04	INR B	05	DCR B
0C	INR C	0D	DCR C
14	INR D	15	DCR D
1C	INR E	1D	DCR E
24	INR H	25	DCR H
2C	INR L	2D	DCR L
34	INR [HL]	35	DCR [HL]
3C	INR A	3D	DCR A

单字节指令 INR和DCR可影响除 CF外的所有标志位。

8080也包含4个循环移位指令，这些指令可使累加器中的内容左移或右移 1位：

操作码	指令	含义
07	RLC	累加器循环左移
0F	RRC	累加器循环右移
17	RAL	累加器带进位循环左移
1F	RAR	累加器带进位循环右移

这些指令只影响 CF。

假定累加器中的值为 A7h，即二进制的 10100111。RLC 指令使 A 中的内容向左移位，最高位（移出顶端）成为最低位（移进底端），同时决定进位标志位 CF 的状态。其结果为 01001111 且 CF = 1。RRC 指令用同样的方法向右移位。开始为 10100111，执行 RRC 指令后，其结果为

11010011且CF = 1。

RAL和**RAR**指令有些不同。当向左移位时，**RAL**指令把 **CF**移入累加器的最低位，而把最高位移入 **CF**中。例如，如果累加器的内容为 10100111，**CF** = 0，**RAL**指令执行的结果是累加器的内容变为 01001110，且**CF** = 1。同样，在相同的初始条件下，**RAR**指令使累加器的内容变为01010011，**CF** = 1。

对于乘2（左移1位）和除2（右移一位）操作，移位指令非常方便。把微处理器寻址的存储器叫作随机访问存储器（**RAM**）是有原因的：微处理器可以简单

地根据提供的地址访问某一存储位置。**RAM**就像一本书一样，我们可以打开它的任何一页。它并不像做在微缩胶片上的一个星期的报纸，要找到周六版，需扫过大半周。同样，它也不

同于磁带，要播放磁带上的最后一首歌需快进整个一面。微缩胶片和磁带的存储不是随机访

问的，而是顺序访问的。

RAM确实效果不错，对于微处理器来说更是如此。但在使用存储器时有所差别是有好处的，下面就是一种既非随机又非顺序访问的存储方式：假定你在一个办公室里，人们到你桌前给你分配工作，每个工作都需要某种文件夹。通常你会发现你在继续某项工作之前，必须使用另外一个文件夹先做一些相关的工作。因此你把第一个文件夹放在桌子上，又拿出第二个文件夹放在它上面进行工作。现在又有一个人来让你做一个优先权高于前面工作的工作，你拿来一个新文件夹放在那两个上面继续工作。而此项工作又需要另外一个文件夹，这样在你的桌子上很快就摆了一堆文件夹了。

注意，这个堆非常明确地、有序地保存了你正在做的工作的轨迹。最上面的文件夹总是 最高优先权的工作，去掉这个以后，下一个肯定是你就要做的，如此类推。当你最终去掉了 桌子上的最后一个文件夹后（你开始的第 1 项工作），你就可以回家了。

以这种方式工作的存储器技术叫做作堆栈（**stack**）。从底向上压入堆栈，从顶向下弹出堆 栈，因此这也叫后进先出存储器，或 **LIFO**。最后放入堆栈中的数据最先被取出，最先放入堆 栈中的数据最后被取出。

计算机中也可以使用堆栈，不是用来保存工作而是用来存储数据，且已被证明使用起来

非常方便。向堆栈中放入数据叫作 **push**（压入），从堆栈中取走数据叫作 **pop**（弹出）。假定你正在用汇编语言设计程序，程序中使用寄存器 **A**、**B** 和 **C**。但在编程过程中，你

发现此程序需要去做另一件事——一个小的计算，其中也要使用寄存器 **A**、**B**、**C**。而你最终 要回到先前的程序，并使用 **A**、**B**、**C** 中原有的值。

当然，你能做的工作只是简单地把寄存器 **A**、**B**、**C** 中的值保存到存储器中的不同位置，

以后再把这些位置的值装载到寄存器中，但这样做需要保存值被保存的位置。一个显然的方 法是把寄存器压入堆栈：

PUSH A

PUSH B

PUSH C

一会儿再解释这些指令的作用。现在，我们只需要知道它们以某种方式把寄存器的内容 保存在一个后进先出的存储器中。一旦这些语句执行了，你的程序就可以毫无顾虑地利用这 些寄存器来做其他工作。为了得到原来的值，只需简单地按与压入堆栈相反的顺序把它们从 堆栈中弹出即可，如下所示：

POP C

POP B

POP A

记住是后进先出。如果用错了 **POP** 语句的顺序，就会引起错误。堆栈机制的一个好处在于一个程序的不同部分都可以使用堆栈而不会出现问题。例如，

在把 **A**、**B**、**C** 压入堆栈中后，程序的其他部分还可能需把寄存器 **C**、**D**、**E** 的内容压入堆栈：

PUSH C

PUSH D

PUSH E

接着，这一部分程序所要做的就是第一部分弹出 **C**、**B** 和 **A** 之前，用下述方法恢复寄存器的值：

POP E

POP D

POP C

堆栈是怎样实现的呢？首先，堆栈只是不被别的东西使用的正常的 **RAM** 的一部分。**8080** 微处理器包含一个特殊的 **16** 位寄存器来对这一部分存储器进行寻址，这个 **16** 位寄存器叫作堆 栈指针。

这里举的压入和弹出寄存器的例子对于 **8080** 来说不太准确。**8080** 的 **PUSH** 指令实际上是存储 **16** 位的值到堆栈，**POP** 指令用来恢复它们。因此 **8080** 不用像 **PUSH C** 和 **POP C** 这样的指令，它有下列 **8** 条指令：

操作码	指令	操作码	指令
C5	PUSH BC	C1	POP BC
D5 E5 F5	PUSH HL PSW	D1 E1 F1	POP DE POP HL POP PSW

PUSH BC 指令把寄存器 **B** 和 **C** 的内容保存到堆栈中，**POP BC** 指令恢复它们。最后一行的缩写 **PSW** 指的是程序状态字，前面讲过，它是包含有标志位的 **8** 位寄存器。最后一行的两条指令实际上是把累加器和 **PSW** 都压入和弹出堆栈。如果你想保存所有寄存器和标志位的内容，可以使用：

PUSH PSW

PUSH BC

PUSH DE

PUSH HL

当以后想恢复这些寄存器的内容时，按相反的顺序使用 **POP**指令：

POP HL

POP DE

POP BC

POP PSW

堆栈是怎样工作的呢？假设堆栈指针为 8000h，**PUSH BC**指令将引起下面这些情况发生：

- 堆栈指针减 1至7FFFH
- 寄存器**B**的内容保存在堆栈指针所指的地址处，即 7FFFh处

- 堆栈指针减 1至7FFEh
- 寄存器C的内容保存在堆栈指针所指的地址处，即 7FFEh处 当堆栈指针仍然为 7FFEh 时，POP BC指令执行，用来反向执行每一步：
- 从堆栈指针所指的地址（即 7FFEh）处装载数据到寄存器 C中
- 堆栈指针增 1至7FFFh
- 从堆栈指针所指的地址（即 7FFFh）处装载数据到寄存器 B中
- 堆栈指针增 1至8000h

对每个 PUSH指令，堆栈都会增加 2个字节，这可能导致程序出现小毛病 —堆栈可能会变得很大以致会覆盖掉程序所需的一些代码和数据。这就是堆栈上溢问题。同样，过多的 POP指令会过早用光堆栈内容，这就是堆栈下溢问题。

如果8080同一个64KB的存储器连接，你可能想把初始堆栈指针置为 0000h。第一条 PUSH 指令使地址减 1变为 FFFFh，这时堆栈占用了存储器的最高地址。如果你的程序放在从 0000h 处开始的存储器区域，则它和堆栈离的就太远了。

对堆栈寄存器进行赋值的指令是 LXI，即load extended immediate（装载扩展的立即数）。下面这些操作码后的指令也是把两个字节装载到 16位寄存器：

操作码		指令
01	LXI BC ,	xxxx
11	LXI DE ,	xxxx
21	LXI HL ,	xxxx
31	LXI SP ,	xxxx

指令：

LXI BC , 527Ah

等价于

MVI B , 52 MVI C , 7A h

LXI指令保存一个字节。另外，上表中最后一条 LXI指令用来对堆栈指针赋值。微处理器 复位后，这条指令并不常用来作为首先执行的指令之一：

0000 h : LXI SP , 0000 h

也可以对寄存器对和堆栈指针执行加 1和减1操作，就好像它们是 16位寄存器一样：

操作码	指令	操作码	指令
03	INX BC	0B	DCX BC
13	INX DE	1B	DCX DE
23	INX HL	2B	DCX HL
33	INX SP	3B	DCX SP

即然是在讨论 16位指令，可以看看更多一些这样的指令。下面的指令是把 16位寄存器对 的内容加到寄存器对 HL中：

操作码	指令
09	DAD HL , BC
19	DAD HL , DE
29	DAD HL , HL
39	DAD HL , SP

上面这些指令可节约几个字节。例如，第一条指令正常需要 6 个字节：

```
MOV      A ,  L

ADD      A ,  C

MOV      L ,  A

MOV      A ,  H

ADC      A ,  B

MOV      H ,  A
```

DAD指令通常用于计算存储器地址，这条指令只影响 CF。下一步让我们看以下各种指令。下面的两个操作码后都紧跟着一个 2 字节地址，分别保存

和装载寄存器对 HL 的内容：

操作码	指令	含 义
2h 2Ah	SHLD [aaaa] , HL LHLD HL , [aaaa]	直接 保存 HL 直接 装载 HL

寄存器 L 的内容保存在地址 aaaa 处，寄存器 H 的内容保存在地址 aaaa+1 处。下面两条指令用来从寄存器对 HL 中装载程序计数器

PC或堆栈指针 SP:

操作码	指令	含义
E9h F9h	PCHL PC , HL SPHL SP , HL	把 HL 中的 内容 装载 到 PC 把 HL 中的 内容 装载 到 SP

PCHL指令实际上是一种转移指令， 8080执行的下一条指令保存在 HL 寄存器对中的地址 所对应的存储单元中。 SPHL是另外一个设置 SP的方法。

下面两条指令中，第一条指令使 HL的内容与堆栈中最上面的两个字节进行交换， 第二条

指令使HL的内容与寄存器对 DE的内容进行交换：

操作码	指令	含义
E3h EBh	XTHL HL , [SP] XCHG HL , DE	HL 与堆 栈顶 端的 内容 交换 DE 和 HL 交换

除了PCHL外，还没有讲过 8080的转移指令。前面第 17章中讲过，处理器中有一个叫作程序计数器 PC的寄存器，PC中包含处理器取回并执行的指令的存储器地址。通常 PC使处理器顺序执行存储器中的指令，但有些指令 — 通常命名为 Jump（转移）、Branch（分支）或 goto

（跳转） — 能使处理器偏离这个固定的过程。这些指令使得 PC装载另外的值，处理器所取的

下一条指令将在存储器的其他位置。尽管简单、普通的转移指令很有用，但条件转移指令更有用。这些指令可使处理器根据

某些标志，如 CF或ZF，来转移到另外的地址处。条件转移指令的存在使得第 17章中的自动加法机成为一般意义上的数字计算机。

8080有5个标志位，其中 4个对条件转移指令有用处。8080支持9种不同的转移指令，包括

无条件转移指令和基于 ZF、CF、PF、SF是1还是0的条件转移指令。在介绍这些指令之前，先介绍一下与此相关的另外两种指令。第一个是 Call（调用）指令。

Call指令与 Jump指令的不同之处在于：前者把一个新值装入到程序计数器 PC中，处理器保存

PC中原来的地址，保存在哪里？当然，在堆栈中。

这种策略意味着 Call指令可有效地保存“程序从哪里跳转”的标记。处理器最终可利用此

地址返回到原来的位置。这个返回指令叫 **Return**。**Return**指令从堆栈中弹出两个字节，并把该值装载到 **PC**中。

Call和**Return**指令是任何处理器中都很重要的功能。它们允许程序员编写子程序，子程序是程序中经常用到的代码段。（“经常”一般意味着“不止一次”。）子程序是汇编语言中的基本组成部分。

让我们看一个例子。假设你正在编写一个汇编语言程序，并且需要使两个数相乘，因此你可以写出一段代码来做这项工作，然后继续往下编写程序，现在又需要使两个数相乘。因为你已知道如何进行两数相乘，因此你只需简单地重复使用同样的指令来完成它。但只是简单地两次把这些指令输入到存储器吗？希望不是，这是对时间和存储空间的浪费，更好的方法是转送到原来的代码处。由于无法返回到程序的当前位置，所以一般的 **Jump**指令不能用。但使用**Call**和**Return**指令可以让你完成所需的功能。

进行两数相乘的一组指令可以作为一个子程序。下面就是这样的子程序。在第 17章中，被乘数（和结果）存放在存储器的某一地址中；而在 8080子程序中，寄存器 **B**的值和寄存器 **C** 中的值相乘，然后把 16位乘积装入寄存器 **HL**中：

```
Multiply:      PUSH PSW      ; 保存要改变的寄存器

               PUSH BC

               SUB H, H       ; 设置 HL（结果）为 0000h
               SUB L, L

               MOV A, B       ; 乘数送到 A

               CPI A, 00h     ; 如果为 0，结束

               JZ AllDone
```

```

MVI B,00h          ; BC 的高字节置 0

Multloop:          DAD HL,BC          ; BC 加到 HL DEC A          ; 乘数减 1 JNZ Multloop          ; 不为 0, 转移

AllDone:           POP BC            ; 恢复保存的寄存器

POP PSW

RET                ; 返回

```

注意，上述子程序的第 1 行开始有一个标号 **Multiply**。当然，这个标号对应于子程序所在的存储器地址。子程序开始用了两个 **PUSH** 指令，通常在子程序开始处应先保存（以后恢复）它需要使用的寄存器。

然后该子程序把 **H** 和 **L** 寄存器置为 0。虽然可以使用 **MVI** 指令而不用 **SUB** 指令，但那需要使用 4 个字节的指令而不是 2 个字节的指令。子程序执行完后，寄存器对 **HL** 中保存有相乘的结果。下一步该子程序把寄存器 **B** 的内容（乘数）移入 **A** 中，并且检查它是否为 0。如果它为 0，

乘法子程序到此结束，因为结果为 0。由于寄存器 **H** 和 **L** 已经为 0，因而子程序可以只使用 **JZ** 指令跳转到末端的两个 **POP** 指令处。

否则，子程序把寄存器 **B** 置为 0。现在，寄存器对 **BC** 中包含一个 16 位的被乘数，**A** 中为乘数。**DAD** 指令把 **BC**（被乘数）加到 **HL**（结果）中。**A** 中的乘数减 1，且只要它不为 0，**JNZ** 指令就又使 **BC** 加到 **HL** 中。此小循环继续下去，直到 **BC** 加到 **HL** 中的次数等于乘数。（可以用 8080 的移位指令编写一个更有效的乘法子程序。）

利用这个子程序完成 25h与12h相乘的程序用下面的代码：

```
MOV B,      25h
```

```
MOV C,      12h
```

```
CALL Multiply
```

Call指令把 **PC**的值保存在堆栈中，该值是 **Call**指令的下一条指令的地址。然后， **Call**指令 使程序转移到标号 **Multiply**所标识的指令，即子程序的开始。当子程序计算完结果后，执行 **RET**（返回）指令，即从堆栈中弹出程序计数器的值，程序继续执行 **Call**指令后面的语句。

8080指令集中包括条件 **CALL**指令和条件 **Return**指令，但它们远不如条件转移指令用得多了。下表中完整地列出了这些指令：



条件 操作码 指令

操作码 指令

操作码 指令

你可能知道，存储器并不是唯一连接在微处理器上的设备。一个计算机系统通常需要输入输出设备以便于实现人机通信。输入输出设备通常包括键盘和显示器。

微处理器是怎样与外围设备 (对于连接到微处理器而不是存储器的东西的称呼) 进行通信的

呢？外围设备具有与存储器相似的接口，微处理器可通过对应于外设的具体地址来对外设进行读写。在有些微处理器中，外围设备实际上占用了通常用来寻址存储器的地址，这种配置叫作内存映像 I/O。然而在 8080 中，在 65 536 个正常地址外还有 256 个附加地址专门为输入输出设备预留，这些就是 I/O 端口 (I/O Port)。I/O 地址信号为 $A_{16} \sim A_{20}$ ，但 I/O 访问与存储器访问

0 7

不同，由 8228 系统控制芯片锁存的信号来区分。

OUT 指令用于把累加器中的数据写到紧跟该指令的字节所寻址的 I/O 端口中。IN 指令把端口的数据读入到累加器中。

操作码 指令

D3 OUT PP

DB IN PP

外围设备有时需要引起微处理器的注意。例如，当你在键盘上按键时，如果微处理器能马上知道这件事通常是有帮助的。这由称作中断 (interrupt) 的机制来完成，这是连接至

8080INT输入端的，由外设产生的信号。

然而，当 8080复位时，它不能对中断产生响应。程序必须通过执行 EI（Enable interrupts）指令来允许中断，通过执行 DI（Disable Interrupts）指令来禁止中断。

操作码	指令
F3	DI
FB	EI

8080的INTE输出端信号表明允许中断。当外设需要中断微处理器当前工作时，它把8080的INT输入端设置为 1。8080通过从存储器中取出指令对它作出响应，但控制信号表明有中断 发生。外设通常通过提供下述指令之一来响应 8080:

操作码	指令		操作码	指令
C7	RST	0	E7	RST 4
CF	RST	1	EF	RST 5
C7	RST	2	E7	RST 6
DF	RST	3	FF	RST 7

以上称作 Restart指令，它们与 CALL指令相似，也需要把当前程序计数器的值压入堆栈。但Restart指令随后转移到一个特定的位置： RST 0 转移到地址 0000h 处， RST 1 转移到地址 0008h处等等，直到 RST 7转移到地址 0038h处。位于这些地址中的代码段来处理中断。例如， 来自键盘的中断引起 RST 4 指令执行，地址 0020h处的一些代码从键盘读取数据（这将在第 21 章做全面介绍）。

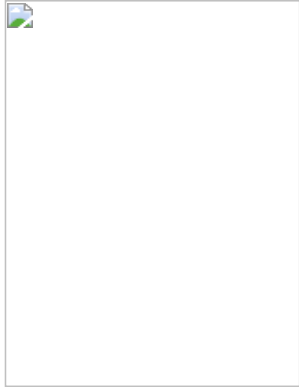
到此为止，已讲述了 243个操作码。下述 12个字节与任何操作码无关： 08h、10h、18h、20h、28h、30h、38h、CBh、D9h、DDh、EDh和FDh。这样总共有 255个操作码。下面还要 提到一个操作码:

操作码	指令
-----	----

NOP代表 no op, 即no operation（无操作）。NOP指令使微处理器什么都不做。这有什么

作用吗？用于填空。8080通常可以执行一批 NOP指令而不会有任何坏情况发生。以下不打算再详细讨论 Motorola 6800，因为它的设计与功能与 8080非常相似。下面是

6800的40个管脚：



代表接地， V

V

SS

是5V电源。与 8080相似， 6800有16个地址输出信号和既可作为输入又

可作为输出的 8 个数据信号。它有 RESET 信号和 R/

W信号。

IRQ信号代表中断请求。6800的

时钟信号比 8080 的更加简单。6800 没有 I/O 端口的概念，所有输入输出设备都必须是 6800 存储器地址空间的一部分。

6800 有一个 16 位程序计数器 PC、一个 16 位堆栈指针 SP、一个 8 位状态寄存器（作为标志）以及两个 8 位累加器 A 和 B。它们都被看成是累加器（B 不是只作为一个寄存器）是因为没有能用 A 来做而不能用 B 来做的事。6800 没有附加的 8 位寄存器。

6800 中有一个 16 位索引寄存器（index register），可用来保存一个 16 位地址，很像 8080 中的寄存器对 HL。对于许多指令来说，它们的地址都可以由索引寄存器和紧跟在操作码后的地址之和得到。

虽然 6800 和 8080 所实现的操作相同——装载、保存、加法、减法、移位、转移、调用，但很明显的区别是：它们的操作码和助记符完全不同。例如，下面是 6800 的分支转移指令：

操作码	指令	含义
20h	BRA	转移
22h	BHI	大于则转移
23h	BLS	小于或相同则转移
24h	BCC	进位为 0 则转移

25h	BCS	进位置1则转移
26h	BNE	不等则转移
27h	BEQ	相等则转移
28h	BVC	溢出为0则转移
29h	BVS	溢出置1则转移
2Ah	BPL	为正则转移
2Bh	BMI	为负则转移
2Ch	BGE	大于或等于0则转移
2Dh	BLT	小于0则

2Eh

BGT

转移

大于
0 则转移

2Fh

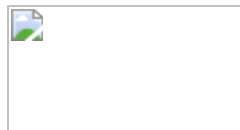
BLE

小于
或等于
0 则转移

6800没有像 8080中那样的奇偶标志位 PF，但它有一个 8080中没有的标志位—溢出标志位（overflow flag）。上述转移指令中有些依赖于标志位的组合。

当然8080和6800指令集是不同的，这两个芯片是同一时间由不同的两个公司的两组不同的工程师设计的。这种不兼容性意味着每一种芯片不能执行对方的机器代码，为一种芯片开发的汇编语言程序也不能翻译成可在另一种芯片上执行的操作码。编写可在多于一种处理器上执行的计算机程序是第 24章的主题。

8080和6800还有一个有趣的不同点：在两种微处理器中，LDA指令都是从一个特定的地址处装载到累加器。例如，在 8080中，下列字节序列：



8080LDA 指令

将把存储在地址 347Bh处的字节装载到累加器。现在把上述指令与 6800的LDA指令相比较，后者采用称作 6800的扩展地址模式：



6800LDA 指令

该字节序列把存储在地址 7B34h 处的字节装载到累加器 A中。这种不同点是很微妙的。当然，你也可能认为它们的操作码不同：对 8080来说是3Ah，对

6800来说是 B6h。但主要是这两种微处理器处理紧随操作码后的地址是不同的，8080认为低位在前，高位在后；6800则认为高位在前，低位在后。

这种 Intel 和 Motorola 微处理器保存多字节数时的根本不同从没有得到解决。直到现在，Intel 微处理器在保存多字节数时，仍是最低有效字节在前（即在最低存储地址处）；而 Motorola 微处理器在保存多字节数时，仍是最高有效字节在前。

这两种方法分别叫作 little-endian (Intel 方式) 和 big-endian (Motorola 方式)。辩论这两种方式的优劣可能是很有趣的，不过在此之前，要知道术语 big-endian 来自于 Jonathan Swift 的

《Gulliver's Travels》，指的是 Lilliput 和 Blefuscu 在每次吃鸡蛋前都要互相碰一下。这种辩论可能是无目的的。先不说哪种方法在本质上说是不是正确的，但这种差别的确当在基于 little-endian 和 big-endian 机器的系统之间共享信息时会带来附加的兼容性问题。

这两种微处理器后来怎样了呢？8080 用在一些人所谓的第一台个人计算机上，不过可能更准确的说法是第一台家用计算机上。下图是 Altair 8800，出现在 1975 年 1 月份的《Popular Electronics》杂志的封面上。



当你看到 Altair 8800 时，前面面板上的灯泡和开关看起来似乎很熟悉。这和第 16 章为

64KB RAM 阵列建议的初始“控制面板”的界面是同一类型的。

8080之后出现了 Intel 8085，更具意义的是出现了 Zilog制造的 Z-80芯片。Zilog是Intel 公司的竞争对手，是由 Intel公司的前雇员，也曾在 4004芯片上做出重要贡献的 Federico Faggin

建立的。Z-80与8080完全兼容，且增加了许多很有用的指令。1977年，Z-80用于Radio Shack TRS-80 Model I上。

也是在1977年，由Steven Jobs 和Stephen Wozniak建立的苹果计算机公司推出了 APPLE II。APPLE II 既不用 8080也不用 6800，而是使用了采用 MOS技术的更便宜的 6502芯片，它是对 6800的增强。

1978年6月，Intel公司推出了 8086，一个 16位微处理器，它可访问的存储空间达到 1MB。

8086的操作码与 8080不兼容，但它包含乘法和除法指令。一年后，Intel公司又推出了 8088，其内部结构与 8086相同，但其外部按字节访问存储器，因此该微处理器可使用较流行的为 8080设计的8位支持芯片。IBM在其5150个人计算机——通常叫作 IBM PC——上使用了 8088芯片，这种个人计算机在 1981年秋季推出。

IBM进军PC市场产生了巨大影响，许多公司都发布了与 PC兼容的机器（兼容的含义在随后各章里将要详细讨论）。多年来，“IBM PC兼容机”也暗指“Intel inside”，特指所谓x86家族的Intel微处理器。Intel x86家族继续发展，1985年出现了32位的386芯片，1989年出现了 486 芯片。1993年初，出现了 Intel Pentium微处理器，普遍地用在 PC兼容机上。虽然这些 Intel微处理器都不断增加了指令的指令集，但它们仍然支持从 8086开始的所有以前处理器的操作码。

苹果公司的 Macintosh 首次发布于 1984年，它使用了 Motorola 68000——一个16位的微处理器，也即 6800的下一代处理器。68000和它的后代（常称为 68K系列）是制造出的最受欢迎的一类微处理器。

从1994年开始，Macintosh计算机开始使用 Power PC，一种由 Motorola、IBM和Apple公司联合开发的微处理器。PowerPC是由

一种称作 **RISC**（精简指令集计算机）的微处理器体系结构来设计的，它试图通过简化某些方面以提高处理器的速度。在 **RISC** 计算机中，每条指令通常长度相同，（在 **PowerPC** 中为 32 位），存储器访问只限于装载和保存指令，且指令做简单操作而不是复杂操作。**RISC** 处理器通常有大量的寄存器以避免频繁访问存储器。

因为 **PowerPC** 具有完全不同的指令集，所以它不能执行 68K 的代码。但现在用于 **APPLE Macintosh** 的 **PowerPC** 微处理器可仿真 68K。运行于 **PowerPC** 上的仿真程序逐个检验 68K 程序的每一个操作码，并执行适当的操作。仿真程序不如 **PowerPC** 自身代码那样快，但可以工作。按照摩尔定律，微处理器中的晶体管数量应该每 18 个月翻一番。这些多增加的晶体管有

什么用处呢？

有些晶体管用于增加处理器的数据宽度，从 4 位到 8 位到 16 位再到 32 位；另外一些增加的晶体管用于处理新的指令。现在大多数微处理器都有用于浮点算术运算的指令（这将在第 23 章解释）；还有一些新增加的指令用来进行一些重复计算，以便在计算机屏幕上显示图片和电影。现代处理器使用了一些技术来提高速度，其中之一是流水线技术，处理器在执行一条

指令的同时读取下一条指令。由于转移指令会改变执行流程，实际上这样达不到预期效果。现在的处理器还包含一个 **Cache**（高速缓冲存储器），它是做在处理器内部的快速 **RAM** 阵列，用于保存最近执行的指令。因为计算机程序经常执行一些小的指令循环，因而 **Cache** 可以避免这些指令重复装载。所有这些速度提升措施都需要在处理器中有更多的逻辑器件和晶体管。

如前所述，微处理器只是完整的计算机系统的一部分（尽管是最重要的部分）。我们将在第 21 章构造这样一个系统，但首先必须学习怎样编码存在存储器中的除了操作码和数字外的其他东西。我们必须返回到小学一年级，再学习一下怎样读写文本。

第 20 章 ASCII 码和字符映射

数字计算机存储器按位存储，所以，需要在计算机上处理的信息必须按位的形式存储。我们已经知道如何用位来表示数和机器码，下一个问题是如何用它来表示文本。毕竟世界上大量堆积的信息是文本形式的，就像装满图书馆的书、杂志和报纸。尽管我们最终要用计算机来存放声音、图像和电影信息，但我们还是以较容易的文本存放开始。

为了以数字形式表示文本，必须开发一些系统使得系统里的每一个字母有唯一的编码。文本中也存在数字和标点符号，所以也必须要有它们的编码。简单地说，所有的字母、数字和符号都要编码，这样的系统叫作字符编码集，每一个编码叫作字符编码。

第一个问题是：这些编码需要多少位？这并不是容易回答的问题。当考虑用位表示文本的时候，需要切合实际。我们习惯于看到书中、报刊、杂志上精美

的文本格式，段落按照相同的间隔整齐地分成一行一行，但这些并不是文本的本质。当我们在杂志上看到一个小故事，几年后在一本书中又看到同样故事的时候，我们不会因为书中文本间距的不同而认为是不同的故事。

换句话说，不要以这种印刷成行列的二维格式来看待文本，应该把文本看成是一维的字母、数字和标点符号流，此外，也许还有额外的编码用来表示一段的结束和另一段的开始。

再来看看，如果在杂志上看到一个故事，后来又在书中看到同样的故事但字样有些不同，这是一个大问题吗？如果杂志上的写法为

而书中的写法为

Call me Ishmael

Call me Ishmael

这些差别难道是我们真正关心的吗？恐怕不是。印刷样式是微妙地影响了文本的观感，但故事本身并没有因为样式的改变而不同。样式可以经常修改，但不会带来什么影响。

接下来另外一个简化问题的方法是：用平版的文本。没有斜体，没有粗体，没有下划线，

没有颜色，没有空心体，没有上下标，没有音调标记，没有 Å、
é、ñ、ö

99%英语文本里纯粹的拉丁字母。

等符号，只有

在对摩尔斯电码和布莱叶盲文的早期研究中，可以看到如何将字母字符表示成二进制的形式。尽管这些系统在特定的场合应用地很好，但用到计算机里都有一些问题。例如：摩尔斯电码是宽度可变的编码：对常用的字符采用短编码，对不常用的字符采用长编码。这样的编码系统适用于电报，但对计算机来说却不合适。另外，摩尔斯电码对字母的大小写没有区分。

布莱叶盲文是宽度固定的编码，很适合计算机。每一个字符由 6 位表示，也可以区分大小写，尽管它是用特殊的 **escape** 码来区分的，该代码表明下一个字符为大写。这也就是说，每个首部字符需要两个代码而不是一个。数字用 **shift** 码表示，在这个特定的代码后紧跟的代码被看作表示数字，直到又一个 **shift** 码将其转换到字符状态。

我们的目标是开发一个字符编码集，使得像如下的句子

I have 27 sisters。

可以用一串代码来表示，每一个代码具有一定的位数。一些代码用来表示字母，一些表示标点符号，一些表示数字。甚至有代码来表示字间的空格。上面的句子中有 18 个字符（包括字间空格），这样一个句子的连续字符代码常称作文本串。

在文本串里，用代码来表示数字 (如 27) 似乎很奇怪，因为前面许多章里已讲过用位来表示数字。我们可能会用简单的二进制数 10 和 111 作为该句中 2 和 7 的代码，但用在这里是不合适的。该句中，字符 2 和 7 可像英文作品中出现的任何一种字符一样来看待，它们可能具有与它们的实际值毫不相干的字符代码。

也许最经济的字符编码是 5 位编码，它首先用于 1874 年的电报机，是由法国电报服务公司职员 **Emile Baudot** 发明的。他的编码 1877 年被服务公司采纳，后来由 **Donald Murray** 修改并在 1931 年被 **CCITT**，即现在的国际电联 (ITU) 标准化。该编码的正式名称是国际电报字母表 NO.2 或 ITA-2，在美国通常称为 **Baudot**，尽管更科学的叫法为 **Murray** 编码。

在 20 世纪，**Baudot** 经常用于电传打字机。**Baudot** 电传打字机有一个键盘，除了只有 30 个键和一个间隔棒外，有些像打字机。电传打字机的键实际上是转换器，它产生二进制代码并且通过电传打字机的输出电缆一位紧接一位地传送出去。电传打字机也有打印机制，从电传打字机的输入电缆输入的代码触发电磁铁在纸上打印出字符。

由于 **Baudot** 是 5 位编码，所以总共只有 32 个代码，这些代码的十六进制值范围从 00h ~ 1Fh。下表是 32 个代码所对应的字母表中的字符：

十六进制码	Baudet 字符	十六进制码	Baudet 字符
-------	-----------	-------	-----------

00		10	E
01	T	11	Z
02	<i>Carriage Return</i> (回车)	12	D
03	O	13	B
04	<i>Space</i> (空格)	14	S
05	H	15	Y
06	N	16	F
07	M	17	X
08	<i>Line Feed</i> (换行)	18	A
09	L	19	W
0A	R	1A	J
0B	G	1B	<i>Figure Shift</i> (数字转义)
0C	I	1C	U
0D	P	1D	Q
0E	C	1E	K

代码00h没有指定。其余的 31个代码中， 26个指定给字母表中的字符， 5个用斜体字或短 语表示出来了。

代码04h是空格代码，用来分隔不同的字；代码 02h和08h表示回车和换行。这些术语来自于电传打字机。当在电传打字机上打字并且到了一行的末尾时，按下下一个杠杆或按钮来完成两件事情。第一，使打印头回到开始处，以便从纸的左边开始打印下一行，这是回车。第二，移动打印头紧接至刚完成的那一行的下一行，这是换行。在 **Baudot**中，独立的键产生这两个 代码。打印的时候， **Baudot**电传打字机响应这两个代码，完成相应动作。

在**Baudot**系统里，如何表示数字和标点符号呢？这就是代码 1Bh的作用，在表中标识为数字转义。在数字转义代码之后，所有的代码序列被看作是数字或标点符号，直到遇到字符转

义代码(1Fh)再返回到字符状态。下表是数字和标点符号的代码。

十六进制码	Baudot字符	十六进制码	Baudot字符
00		10	3
01	5	11	+
02	<i>Carriage Return</i>	12	<i>Who Are You?</i>
03	9	13	?
04	<i>Space</i>	14	‘
05	#	15	6
06	,	16	\$
07	°	17	/
08	<i>Line Feed</i>	18	-
09)	19	2
0A	4	1A	<i>Bel</i> (响铃)
0B	&	1B	<i>Figure Shift</i>
0C	8	1C	7

0D	0	1D	1
0E	:	1E	(
0F	=	1F	<i>Letter Shift</i>

实际上，ITU没有定义代码 05h、0Bh和16h，而是保留为“国家使用”，这个表里列出的是美国的用法。这些代码在某些欧洲国家语言中用作重音符号。响铃代码用来敲响电传打字机上能听见的铃声；“Who Are You”代码激活一种机制，用它电传打字机能识别自己。

像摩尔斯电码一样，这 5位编码不能区别大、小写。语句

I SPENT \$25 TODAY.

由下面的十六进制数据流来表示：

0C 04 14 0D 10 06 01 04 1B 16 19 01 1F 04 01 03 12 18 15 1B 07 02 08

注意三个转义代码：1Bh在数字的前面，1Fh在数字的后面，最后一部分之前又有 1Bh。该行代码用回车、换行代码来结束。

然而，如果一行两次传送该数据流到电传打印机，将会出现以下情形：

I SPENT \$25 TODAY. 8' 03,5 \$25 TODAY.

这是怎么回事？打印机接收到的上一行的最后一个转义代码是数字代码，所以第二行开始的代码被解释成数字。

类似这样的问题是采用转义代码所产生的典型的令人烦恼的结果。尽管 Baudot是很经济的编码，但人们可能更想采用能唯一表示字符或标点符号且对大、小写进行区分的代码。

如果想确定比 Baudot更好的编码系统需要多少位，只需把各种符号加起来：大小写字母需52个代码，0~9数字需 10个代码，这

已经有 62 个，加上一些标点符号，则超过了 64 个代码，这意味着需要多于 6 位的编码。但是距离 128 个字符数，似乎还有足够的余地。如果超过 128 个字符，则需要 8 位编码。

所以答案应该是 7。如果不用转换代码来区分大、小写，那么英文里应该用 7 位来表示字符。

这些字符编码都是什么呢？当然，我们可以随心所欲地编码。如果打算自己制造计算机且计算机的每一个硬件都由自己制造，自己编程且不把自己所造的计算机去与任何其他计算机连接，则可以构造自己的编码，所要做的就是给每一个字符一个唯一的编码。

但是因为很少有独立制造和使用计算机这种情形发生，所以通常是大家遵循并使用同一

编码。这样制造出来的计算机就可以与其他计算机兼容，并且可以交换文本信息。我们可能也不应该随意编码，例如，当在计算机上处理文本时，如果字母表上的字符是按

顺序进行编码的，则会带来很多好处，其码这样的顺序使得按字母排序和分类更容易一些。幸运的是，我们已经有了这样一个标准，即美国信息交换标准代码，简写为 **ASCII** 码。它

1967 年正式公布，此后一直是计算机工业界最为重要的标准。除了一个大的例外（在后面讲到），可以肯定的是，无论什么时候处理文本，总会以某种方式涉及到 **ASCII** 码。

ASCII 码是 7 位编码，用二进制代码 0000000~1111111,即十六进制代码 00h~7Fh 来表示。让我们来看 **ASCII** 码，但不要从最开始看，因为前 32 个代码比其余代码理解起来要困难一些。从第二批的 32 个代码开始讲起，它包括标点符号和 10 个数字。下表列出了它们的十六进制代码及对应的字符：

十六进制码 ASCII 字符 十六进制码 ASCII 字符

20	space	30	0
21	!	31	1
22	"	32	2
23	#	33	3
24	\$	34	4

25	%	35	5
26	&	36	6
27	'	37	7
28	(38	8
29)	39	9
2A	*	3A	:
2B	+	3B	;
2C	,	3C	<
2D	-	3D	=
2E	.	3E	>
2F	/	3F	?

注意20h是空格符，用来分隔单词和句子。

接下来的 32个代码包括大写字母和一些附加的标点符号。除 @符号和下划线之外，这些 符号在打字机上不经常出现，它们出现在标准的计算机键盘上：

十六进制码	ASCII字符	十六进制码	ASCII字符
40	@	50	P
41	A	51	Q

42	B	52	R
43	C	53	S
44	D	54	T
45	E	55	U
46	F	56	V
47	G	57	W
48	H	58	X
49	I	59	Y
4A	J	5A	Z
4B	K	5B	[
4C	L	5C	\
4D	M	5D]
4E	N	5E	^
4F	O	5F	-

接下来的32个字符包括所有小写字母和一些附加的标点符号，也是在打字机上不常出现的：

十六进制码	ASCII字符	十六进制码	ASCII字符
60	`	70	p
61	a	71	q
62	b	72	r
63	c	73	s
64	d	74	t
65	e	75	u
66	f	76	v
67	g	77	w
68	h	78	x
69	i	79	y
6A	j	7A	z
6B	k	7B	{

6C	l	7C	
6D	m	7D	}
6E	n	7E	~
6F	o		

注意该表中少了与 7Fh对应的最后一个字符。如果你一直在统计，这三个表总共列出了 95 个字符。因为 ASCII码是7位编码，可以有 128个代码，所以还有 33个代码可用。下面简单地讲一下这些代码。

文本串：

Hello, you !

可以表示成 ASCII码的十六进制形式

48 65 6C 6C 6F 2C 20 79 6F 75 21

注意除了字母代码以外，还有逗号（代码 2C）、空格（代码 20）和感叹号（代码 21）。下面是另一短句：

用ASCII码表示为:

I am 12 years old.

49 20 61 6D 20 31 32 20 79 65 61 72 73 20 6F 6C 64 2E

注意句中数字 12 表示成十六进制数 31h 和 32h，分别是数字 1 和 2 的 ASCII 码。当数字 12 作为文本流的一部分时，它不应该被表示成十六进制码 01h 和 02h，或者 BCD 码 12h，或者十六进制代码 0Ch。这些代码在 ASCII 码里都表示的是其他意思。

ASCII 码表示的大写字母与其对应的小写字母的 ASCII 码值相差 20h，这使得编写某些程序代码更为容易，如：把一个字符串变成大写。假设在内存的某个区域存放有字符串，一个字节放一个字符。下面的 8080 子程序假设字符串的首地址存放在寄存器 HL 中；寄存器 C 存放有字符串

的长度，即字符串中的字符个数：

```
Capitalize: MOV A,C          ;C=number of characters left ( C 为余下的字符数) CPI A,00h      ;Compare with 0 (与 0 比较)
```

```
JZ AllDone          ;If C is 0, we're finished (如果 C 为 0，则结束)
```

```
MOV A,[HL]          ;Get the next character (取下一个字符)
```

```
CPI A,61h           ;Check if it's less than 'a' (判断是否小于 'a')
```

```
JC SkipIt      ;If so,ignore it( 如果是, 则跳过 )
```

```
CPI A,7Bh      ;Check if its greater than 'z' (判断是否大于 'z' ) JNC SkipIt      ;If so ,ignore it( 如果是, 则  
跳过 )
```

```
SBI A,20h      ;It' s lowercase,so subtract 20h (是小写, 则减 20h) MOV [HL],A      ;Store the character( 保存  
字符 )
```

```
SkipIt: INX HL      ;Increment the text address (字符地址加 1) DCR C      ;Decrement the counter (计数器减 1)
```

```
JMP Capitalize ;Go back to the top (返回到程序开始处)
```

```
AllDone:      RET
```

从小写字母减去 20h转换成大写字母的语句可以用下面的语句代替:

```
ANI A,DFh
```

ANI指令是一个“与”立即数的操作, 它把累加器中的数值与 **DFh** (即二进制数 **11011111**) 执行按位“与”操作, 即把两个数的对应位进行“与”操作“与”操作保留 **A**中的所有位, 除了从左边数第3位

被置成0。把这个位设置为0也即把ASCII码表示的小写字母转换成大写字母。

上面列出的 95个代码也称作图形字符，因为它们可以显示出来。ASCII码还包括 33个控制

字符，它们不能显示出来但表示执行某一特定功能。鉴于完整性，这里列出了 33个控制字符，即使它们很难理解也不要担心。在 ASCII码公布以后，更多地是想把它们用在电传打字机上，现在许多代码已经很少见到了。

十六进制码	缩写词	控制字符名称
00	NUL	空
01	SOH	标题开始
02	STX	文本开始
03	ETX	文本结束
04	EOT	传输结束
05	ENQ	询问
06	ACK	应答
07	BEL	响铃

08	BS	退格
09	HT	水平制表
0A	LF	换行
0B	VT	垂直制表
0C	FF	换页
0D	CR	回车
0E	SO	移出
0F	SI	移入
10	DLE	转义
11	DC1	设备控制 1
12	DC2	设备控制 2
13	DC3	设备控制 3
14	DC4	设备

控制
4

15

NAK

否定
应答

16

SYN

同步

17

ETB

传输
块
结束

(续)

十六进制 制码	缩写 词	控制字 符名称
	CAN	作废
18		
19	EM	载体 结束
1A	SUB	替代字 符
1B	ESC	扩展
1C	FS	文件分 隔或信 息分隔 4
1D	GS	组分 隔或信 息分隔 3
1E	RS	记录分 隔或信 息分隔 2

1F	US	单元分隔或信息分隔 1
7F	DEL	删除

控制字符可以与图形字符混合使用来设置一些基本的文本格式。这很容易理解，想像一下诸如电传打字机或简单打印机之类的设备，它们对 **ASCII** 码流作出的响应是在纸上打印出字符。设备的打印头通过打印一个字符并向右移动一格来对 **ASCII** 码作出响应。上面这些很重要的控制字符就用来改变这种通常的动作。

例如：看以下的十六进制字符串

41 09 42 09 43 09

09 字符是一个水平制表符，简称 **Tab**。假设打印页面上所有的水平字符位置是从 **0** 开始，**Tab** 的作用是在下一个水平位置即 **8** 的倍数处开始打印下一个字符，如下所示：

A B C

这是保持字符按列对齐的简便方法。

换页符（**12h**）的作用是使打印机跳过当前页开始打印下一页。退格符用来在一些旧的打印机上打印复合字符，例如，假设计算机要控制电传打字机以

重音标记来打印小写字母 **e**，即 **è**，可以通过用十六进制码 **65 08 60** 来实现。最重要的控制字符是回车和换行，它们与 **Baudot** 码中的回车换行符意义相同。在打印机

中，回车符使打印头移到打印页面的左边，换行符使打印头移到下一行，用两个代码通常表示从新的一行开始。单独使用回车符可以用来在一个已有的行上打印，单独使用换行符可以用来跳到当前位置的下一行而不移到左边。

尽管 **ASCII** 码是计算机世界的主要标准，但在许多 **IBM** 大型机系统上却没有采用。在 **System/360** 系统中，**IBM** 研制了自己的 **8** 位字符编码，即扩展的 **BCD** 交换代码 **EBCDIC**。该编码是早期的 **BCDIC** **6** 位编码的扩展，从 **IBM** 穿孔卡片使用的代码演变而来。穿孔卡片可以存放 **80** 个文本字符，这种模式由 **IBM** **1928** 年首先引入并且用了 **50** 多年。



当考察穿孔卡片与相关的 8 位EBCDIC字符编码的关系时，需要记住的是，在若干种不同 技术的支持下这种代码已经经历了好几代的演变。正因为如此，不要指望从中发现太多的逻辑性和一致性。

穿孔卡片中，字符编码由一列上穿出的一个或多个矩形孔的组合而形成，字符本身通常 在接近卡片的上边沿处打印出来。下面的 10行由数字标识，分别是第 0行、第 1行直到第 9行。

在第0行之上没有数字的行为第 11行，最上边的行为第 12行，没有第 10行。以下是一些常用的 IBM穿孔卡片术语：行 0～9称作数字行或数字穿孔，行 11和12称作区

域行或区域穿孔。有一些 IBM穿孔卡片也会带来混淆，把行 0和9看作是区域行而不是数字行。一个8位EBCDIC字符编码由高半字节（4位）与低半字节组成。低半字节是与字符的数字

穿孔一致的 BCD码，高半字节是与区域穿孔一致的编码。回忆一下第 19章的 BCD编码标准，

它是用二进制编码十进制数，即用 4位编码来代表十进制的 0～9。

对数字 0～9，没有区域穿孔，没有区域穿孔对应的高半字节是 1111，低半字节是数字穿 孔的BCD码。下面是 0～9的EBCDIC编码：

十六进制码	EBCDIC字符
F0	0
F1	1
F2	2
F3	3
F4	4

F5	5
F6	6
F7	7
F8	8
F9	9

对大写字母，如果只有第 12行有穿孔，则用 1100来标识；如果只有第 11行有穿孔，则用

1101来标识；如果只有第 0行有穿孔，则用 1110来标识。大写字母的 EBCDIC编码为：

十六进制码 EBCDIC字符		十六进制码 EBCDIC字符		十六进制码 EBCDIC字符	
C1	A	D1	J	E2	S
C2	B	D2	K		
C3	C	D3	L	E3	T
C4	D	D4	M	E4	U
C5	E	D5	N	E5	V
C6	F	D6	O	E6	W

C7	G	D7	P	E7	X
C8	H	D8	Q	E8	Y
C9	I	D9	R	E9	Z

注意这些编码的编号次序。在一些场合，当用 EBCDIC 文本编写程序的时候，这些次序有时还真令人头痛。

小写字母与大写字母的数字穿孔相同但区域穿孔不同。小写 a~i 的第 12 行和第 0 行有穿孔，相应的区域代码为 1000；j~r 的第 12 行和第 11 行有穿孔，区域代码为 1001；s~z 的第 11 行和第

0 行有穿孔，区域代码为 1010。小写字母的 EBCDIC 编码为：

十六进制码	EBCDIC字符	十六进制码	EBCDIC字符	十六进制码	EBCDIC字符
81	a	91	j		
82	b	92	k	A2	s
83	c	93	l	A3	t
84	d	94	m	A4	u
85	e	95	n	A5	v
86	f	96	o	A6	w
87	g	97	p	A7	x
88	h	98	q	A8	y
89	i	99	r	A9	z

当然，标点符号和控制字符也有 **EBCDIC** 编码，但对该编码系统的全面考察并不需要。似乎 **IBM** 穿孔卡片上的每一列就足以提供 12 位的编码信息，每个孔代表 1 位，是这样吗？

果真如此的话，可以用穿孔卡片上每一列 12 个位置中的 7 个来表示 ASCII 码的字符代码。但是，实际上，这并不合适，太多的穿孔将会影响到卡片物理状态的平直。

EBCDIC 中的许多 8 位码没有定义，建议采用 ASCII 码的 7 位编码是合理的。当 ASCII 码研制出来的时候，存储器非常昂贵，于是有些人感到 ASCII 码应该用 6 位码并且采用转义字符来区分大小写用以节约存储器。这种观点没有被接受，相反，人们认为 ASCII 码应该是 8 位编码，因为即使在当时人们也普遍认为计算机应该是按 8 位存储，而不是 7 位。当然，8 位字节现在是标准的。因此，尽管 ASCII 码在技术上是 7 位编码，但它普遍是按 8 位值来存储的。

字节与字符之间的等价关系的确很方便，我们只需简单地通过统计字符数就可以粗略估计一个文本文件所需要的存储空间。当然，用 K 和 M 来表示计算机存储空间用得更为广泛一些。

例如，传统的 8.5×11 英寸的双倍空隙打印页面有 1 英寸的页边空白和大约 27 行的正文。每行约 6.5 英寸宽，每英寸有 10 个字符，这样一页共有 1750 个字节。单倍空隙打印页面大约是它的 2 倍，约 3.5KB。

《NEW Yorker》杂志每页有 3 列，每列有 60 行，每行大约有 40 个字符，这样每页有 7200 个字符（或字节）。《纽约时代》每页有 6 列。如果页面都是文字而没有标题和图片（这是不常有的），则每列有 155 行，每行大约有 35 个字符，从而整个页面有 32 550 个字符，即 32KB。

硬面书每页大约 500 个字，平均每个字有 5 个字母，确切的说，应该是 6 个字符，因为要把字间空格统计在内，这样书的每页约 3000 个字符。假设平均每本书有 333 页（这可能是一个估计较高的数字），则意味着平均每本书大约是 1MB。

当然，每本书的差别很大：

F. Scott Fitzgerald 的《The Great Gatsby》大约 300KB。J. D. Salinger 的《Catcher in the Rye》大约 400KB。

Mark Twain 的《The Adventures of Huckleberry Finn》大约 540KB。John Steinbeck 的《The Grapes of Wrath》大约 1MB。

Herman Melville 的《 Moby Dick 》大约 1.3MB 。

Henry Fielding 的《 The History of Tom Jones 》大约 2.25MB 。 Margaret Mitchell 的《 Gone With the Wind 》大约 2.5MB 。

Stephen King 的《 The Stand 》大约 2.7MB 。

Leo Tolstoy 的《 war and Peace 》大约 3.9MB 。

Marcel Proust 的《 Remembrance of Things Past 》大约 7.7MB 。

美国国会图书馆大约有 20 000万本书，总共有 20万亿字符，即 20TB的文本数据。（图书馆 还有大量的照片和录音。）

尽管ASCII码是计算机世界里最重要的标准，但它并不是完美的。ASCII码的最大问题在于它太倾向于美国！的确，ASCII码即使对那些以英语为主要语言的国家也几乎是不合适的。尽管ASCII码包含有美元符号，但英镑符号呢？还有许多西欧国家语言中用到的重音符号呢？更不用说在欧洲一些国家里使用的非拉丁字母，包括希腊文、阿拉伯文、希伯来文和西里尔文。此外，还有印度及东南亚国家用到的婆罗门教的手迹。而一个 7位编码又如何来处理成千上万的中文、日文、韩文笔画以及韩语音节？

在研究 ASCII码的时候，也一直在考虑其他国家的需要，尽管没有充分考虑非拉丁字母。根据公开的 ASCII码标准，10个ASCII码代码（40h、5Bh、5Ch、5Dh、5Eh、60h、7Bh、7Ch、7Dh和 7Eh）可用来重新定义而为某一国家使用。另外，如果需要，数字符号（#）可用英镑符号（£）替换，美元符号（\$）可用通用货币符号（¤）替换。显而易见，只有使用包含这些替换符号的特定文本文档的所有人都知道这些变化的时候，替换符号才有意义。

由于许多计算机系统按 8位来存储字符，则可以设计扩展的ASCII码字符集来包含 256个字符而不仅仅是 128个。在这样的字符集里，代码 00h~7Fh定义成与 ASCII码一致；代码 80h~ FFh可定义成表示另外的字符。这种技术已被用来定义附加的字符代码，包含重音字母及非拉丁字母。作为例子，这里有一个 96个字符的 ASCII码的扩展，称之为第 1号拉丁字母表，定义的字符编码从 A0h~FFh。在该表里，十六进制字符编码的高半字节由最高行给出，低半字节由左边列给出：



代码A0h对应的字符为非断开空格。通常计算机处理格式文本是按照行和段来编排的，每一行以空格符号断开，对应的ASCII码为20h。代码A0h用来显示一个空格，但不能用来断开一行。非断开空格可以用在如“WW II”这样的文本中。代码ADh定义成软连字符，该连字符用来分开一个字中间的音节，且只在需要连接被两行分开的一个单词时才使用。

遗憾的是，近几十年来出现了许多不同的ASCII码的扩展，导致了混淆和不兼容性。

ASCII码通过扩展甚至可以编码中文、日文和韩文的笔画。有一个流行的编码叫作Shift-JIS

（日本工业标准），其代码81h~9Fh用来表示2字节字符编码的起始字节。以这种方法，Shift-JIS可编码约6000个额外字符。遗憾的是，Shift-JIS不是使用这种技术的唯一系统。在亚洲，还有三个很流行的双字节字符集。

双字节字符集有许多问题，不兼容性只是其中之一。另一个问题是，一些字符——特别是普通的ASCII码字符——是用1个字节编码来表示的，而成千上万的笔画则是由双字节编码来表示，从而导致使用这样的字符集很困难。

在假定会有一个特定的字符编码系统能适用于世界上所有语言的前提下，1988年，几个主要的计算机公司一起开始研究一种替换ASCII码的编码，称为Unicode。鉴于ASCII码是7位编码，Unicode采用16位编码，每一个字符需要2个字节。这意味着Unicode的字符编码范围从0000h~FFFFh，可以表示65 536个不同字符。对世界上所有可用计算机进行来通信的语言来说，有足够的扩展空间。

Unicode编码不是从零开始的，开始的128个字符编码0000h~007Fh与ASCII码字符一致。Unicode编码00A0h~00FFh也与前面

讲到的对 ASCII码扩展的第 1号拉丁字母表一致。其他世界范围的标准也收编在 Unicode中。

尽管 Unicode对现有的字符编码做了明显改进，但并不能保证它能很快被人们接受。ASCII码和无数的有缺陷的扩展 ASCII码已经在计算机世界中占有一席之地，要把它们逐出计算机世界并不是件容易的事。

对Unicode来说的一个实实在在的问题是，它改变了一个文本字符与 1个字节存储器之间的等效关系。用 ASCII码编码，《The Grapes of Wrath》这本书的大小约为 1M 字节；而用 Unicode编码，约是 2MB，这也算是采用 Unicode编码付出的代价吧。

第 21 章 总线连接

处理器无疑是计算机中最重要的部件，但并不是唯一的部件。一台计算机也需要随机访问存储器（**RAM**）来存放机器码指令以便让处理器执行。计算机还必须有一些方法使这些指令进入**RAM**（输入设备）以及一些方法使程序执行结果得以看见（输出设备）。前面讲过，**RAM**是易失性的，当断电时，它的内容就会丢失。所以计算机中另一个有用的部件是永久存储设备，当计算机断电的时候，它们可以保存代码和数据。

组成一台完整计算机的所有集成电路必须安装在电路板上。在一些小型机器上，所有的集成电路可以安装在一块板上，但更常见的是，不同的部件分开安装在两块或更多的板上，这些板之间通过总线互相通信。简单地说，总线是提供给计算机中每块电路板的数字信号的集合，这些信号可以分为 4 类：

- 地址信号。这些信号由微处理器提供，常用来寻址 **RAM** 单元，也可用来寻址连接到计算机上的其他部件。
- 数据输出信号。也由微处理器提供，用来写入数据到 **RAM** 或其他设备。要仔细推敲输入（**input**）和输出（**output**）的含义。数据输出信号是从微处理器输出，变成 **RAM** 和其他设备的数据输入信号。

- 数据输入信号。是由计算机的其余部分提供，由微处理器读入的信号。数据输入信号通常来自于 **RAM** 的输出，也即表示微处理器读入存储器内容。但是其他部件也提供数据输入信号给处理器。

- 控制信号。由各种各样的信号组成，通常与计算机的特定处理器的控制信号一致。控制信号可来自于微处理器或其他部件传送到微处理器。例如，微处理器可用一个控制信号来指示它要写一些数据到某一存储器地址。

另外，总线给计算机中的各个电路板提供电源。

早期家用计算机流行的一种总线是 **S-100** 总线，该总线 1975 年在第一台家用计算机 **MIT S Altair** 上首先采用。尽管这种总线以 **8080** 微处理器为基础，但后来它也被其他一些处理器，如 **6800** 采用。**S-100** 的电路板的规格是 5.3×10 英寸，电路板的一边有 100 个接头可插在插槽里

（这就是 **S-100** 的来源）。

S-100 计算机有一块较大的板称为母板或主板，上面有若干个（如：12 个）互相连接起来的 **S-100** 总线插槽，这些插槽有时也叫扩展槽，**S-100** 电路板（也叫扩展板）插到插槽里。**8080** 微处理器及支持芯片（第 19 章曾提到过）在此 **S-100** 板上。**RAM** 在另一个或更多的其他 **S-100** 电路板上。

S-100 总线是为 **8080** 芯片设计的，它有 16 个地址信号，8 个数据输入信号，8 个数据输出信号（前面讲过，**8080** 自身是把数据输入、输出信号混合在一起的，由 **8080** 所在电路板上的另一个芯片来把这些信号分开成单独的输入、输出信号）。总线上还有 8 个中断信号，这些信号由那些需要 **CPU** 立即做出响应的部件产生。例如（在本章后面将要讲到），当在键盘上敲一个键时，键盘会产生中断信号，**8080** 执行一个小程序确定是哪一个键并做出相应反应。包含

8080的电路板上通常还有一个芯片称作 **Intel 8214** 优先级中断控制单元，它用来处理这些中断。当中断产生时，该芯片产生一个中断信号给 8080，8080响应中断。该芯片提供 **RST (Restart)** 指令使得微处理器保存当前的程序计数器，并根据具体的中断信号转移到地址 0000h、0008h、0010h、0018h、0020h、0028h、0030h 或 0038h 处去执行。

如果正在设计一个具有新的总线类型的新计算机，你可以选择公开你的总线规范或者保密。

如果一个总线规范是公开的，其他厂商——称为第三方厂商——可以设计并销售与这种总线相配套的扩展板。这些附加的扩展板使得计算机更有用且更令人满意，计算机的大量销售为扩展板提供了更大的市场。这种现象刺激许多小的计算机系统设计者坚持开放体系结构的原则，允许其他厂商生产计算机的外围设备。这样总有一种总线最终可以认为是工业界的标准。标准已成为个人计算机工业的重要组成部分。

最著名的开放式体系结构个人计算机起源于 **IBM PC**。1981年秋季，**IBM** 公开了包括整个计算机完整电路图的 **PC** 机技术参考手册，其中还包括 **IBM** 为它制造的所有扩展板。这个手册是很重要的工具，它使得许多制造商可以生产自己的 **PC** 机扩展板并且事实上产生了 **PC** 机的“克隆”体——兼容 **PC** 机，兼容 **PC** 机与 **IBM PC** 机几乎完全相同且运行相同的软件。

源于 **IBM PC** 的更新换代产品现在已占到桌面计算机系统大约 90% 的份额。尽管 **IBM** 自身只有很少的市场份额，但它毕竟要比最初的 **PC** 机采用专有设计的封闭式体系结构所占的份额要大。苹果公司的 **Macintosh** 机开始就采用封闭式体系结构，根本不考虑开放其体系结构，这当初的决定可以用来解释为什么在目前的桌面计算机市场上 **Macintosh** 只占有不到 10% 的份额。

(记住一点，无论一个计算机系统是在开放体系结构还是封闭体系结构下设计，都不会影响到其他公司开发在该计算机系统上运行的软件。只有那些特定的视频游戏软件开发商才会限制其他公司开发用于他们系统的软件。)

最初的 IBM PC使用Intel 8088 微处理器，可寻址 1M存储空间。尽管 8088处理器内部是 16 位，但在外部按照 8位来寻址存储器。IBM为最初的 PC机设计的总线现在称作 ISA (industry standard architecture, 工业标准体系结构) 总线。扩展板上有一个 62针的插头，信号包括 20个 地址信号，8 个组合 (复用) 数据输入 / 输出信号，6 个中断请求信号和 3 个 DMA (direct memory access, 直接存储器访问) 请求信号。DMA允许存储设备 (本章最后将要讲到) 比采用别的方法更快地进行操作。通常，微处理器处理所有的内存读 / 写操作，但通过 DMA，其他设备可绕过微处理器通过总线直接进行内存读 / 写操作。

在S-100系统里，所有的部件都安装在扩展板上。在 IBM PC机里，微处理器、一些支持芯片及一些 RAM安装在IBM所称的系统板上，系统板也常称作主板或母板。

1984年，IBM推出了 Personal Computer AT(先进技术型个人计算机)，它采用 16位的Intel 80286微处理器，可寻址 16M存储器。IBM保留了已有的总线，但另加了一个 36针的插槽，其中包括新增的 7个地址信号 (尽管只需要 4个)，8个数据输入 / 输出信号，5个中断请求信号和 4 个DMA请求信号。

无论是数据宽度 (从 8位到16位到32位) 还是输出的地址信号数目，当处理器在这些方面的增长超出总线能力时，就需要对总线进行更新换代；当处理器达到较高的速度时，它也会超出总线的能力。早期的总线是为时钟频率是几兆赫而不是几百兆赫的处理器设计的。如果总线的设计不能适应高速传输，则可能引起射频干扰 (RFI)，从而引起收音机或电视机附近

的静态或其他噪声干扰。

1987年，IBM推出了微通道体系结构（micro channel architecture，MCA）总线，这种总线的某些方面 IBM已申请了专利，这样 IBM就可以从其他使用这种总线的公司收到授权费用。可能也正因为如此，MAC总线没有成为工业标准。取而代之的是 1988年9家公司(不包括 IBM) 联合推出的 32位EISA（Extended Industry Standard Architecture,扩展的工业标准体系结构）总线。近年来，Intel公司设计的外围部件互联（peripheral component interconnect，PCI）总线在PC兼容机上已普遍采用。

为理解计算机中各种不同部件是如何工作的，让我们再重新回到 70年代中期较质朴的年代。假想我们正在为 Altair或者为我们自己设计的 8080、6800计算机设计电路板，我们可能打算为计算机设计一些存储器，用一个键盘作输入，一个电视机作输出，此外还有一些方法用来保存关闭计算机电源时存储器中的内容。现在来看一看我们所设计的把这些部件添加到计算机中所用的各种各样的接口。

第16章讲过，RAM阵列有地址输入、数据输入、数据输出信号，并且有一个控制信号用来写入数据到存储器。地址输入信号的个数决定了RAM阵列中可以存放的数值的个数：

$$\text{RAM阵列中数值的个数} = 2^{\text{地址输入信号个数}}$$

数据输入/输出信号的数目表明了存放的数值的位数。70年代中期家用计算机中常用的存储器芯片是 2102：



2102是MOS(metal_oxide semiconductor, 金属氧化物半导体)家族的成员，与 8080、6800 中使用的 MOS技术一样，MOS半导体器件很容易与 TTL芯片连接，但前者通常比 TTL器件的晶体管密度高但速度较慢。

通过统计地址信号（A～A）、数据输出信号（DO）和数据输入信号（DI），可以算出这

种芯片可以存放 1024 位。根据采用的 2102 芯片的类型，访问时间——即从一个地址提供给芯片到数据输出成为有效所需要的时间——从 350~1000 纳秒不等。R/W（读/写）信号在读存

储器时是 1。当向芯片写数据时，R/W 必须在至少为 170~550 纳秒的时间段内为 0，这也取决于所选用的 2102 芯片类型。

特别要提到的是 CS 信号，它表示片选。当信号为 1 时，片子不被选中，意味着片子对 R/W

信号不做响应。然而，CS 信号对芯片还有一种重要作用，我们下面将简单描述一下。

当然，如果要为一个 8 位的微处理器组织存储器，则肯定希望存储器按 8 位而不是 1 位存放。按最少计算，需要把 8 个这样的 2102 芯片连接在一起用来存放整个字节。可以把所有 8 个芯片

对应的地址信号、R/W 和 CS 信号连接起来达到此目的，结果如下图所示：



地址

数据输入 数据输出

这是一个 1024×8的RAM阵列，即 1KB的RAM

从实际观点来看，需要把这些存储芯片安装在一个电路板上。在一块板上可以装多少呢？如果真的把它们紧紧安装在一起，可以在一个 S-100板上安装 64个这样的芯片，提供 8KB 存储器。但是，最好还是宽松地安装，用 32个芯片组成 4KB存储器。连接在一起用来存储完整字节的一组芯片（见上图）称为存储体。一个 4KB存储器由 4个存储体组成，每个存储体有 8个芯片。

像8080、6800这样的 8位微处理器有 16位地址可用来寻址 64KB存储器。如果连接的是有 4

个存储体的 4KB存储器板，则存储器板中 16位地址信号完成的功能如下：



选择存储板 选择存储体 寻址 **RAM**

10个地址信号 $A \sim A$ 直接连到 RAM芯片，地址信号 A

和A 用来选择 4个存储体中的一个,

0 9 10 11

地址信号 A

$\sim A$

用来确定是哪一块存储器板。每个 **4KB**存储器板占据微处理器整个 **64KB**存

存储空间16个不同的 4KB地址空间范围中的一个，它们分别是：

0000h ~ 0FFFh 或

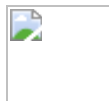
1000h ~ 1FFFh 或

2000h ~ 2FFFh 或

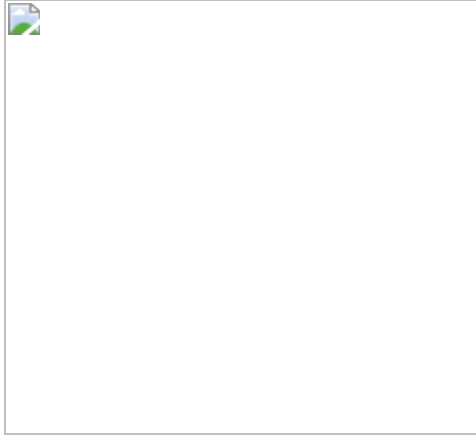
F000h ~ FFFFh

假定一个 4KB存储器板的地址范围是 A000h~AFFFh，则意味着地址 A000h~A3FFh提供 给其中的第 1个1KB存储体，地址 A400h~A7FFh提供给第 2个存储体，地址 A800h~ABFFh提 供给第3个存储体，地址 AC00h~AFFFh提供给第 4个存储体。

连接 4KB存储器板是很经常的，后面可看到如何灵活地确定存储器板的地址范围。为获 得灵活性，可以使用 DIP开关，一种双排直插封装的系列（2~12个）微小开关。它可以插入 到普通的 IC插座中：



可以把这个开关和总线中的高 4位地址信号连接到称作(比较器)的电路中：



前面讲过，只要两个输入中有一个而不是两个都为 1，则XOR（异或）门的输出为 1。这也就是说如果两个输入相同——都为0或都为1——则异或门的输出为 0。

和A

假设闭合对应于 $A \text{ AFFFh}$ 。

的开关，则意味着我们选择的存储器板的地址范围从 A000h～

如果来自总线的地址信号 A

、 A

12

、 A

13

和A

与开关中设置的地址相同，则 4 个异或门的输



出都是0，也即NOR（或非）门的输出为 1:

—

可以把Equal信号通过 2—4译码器生成 CS信号来选择存储器板中的 4个存储体之一：



—

CS 第 1 个存储体

—

CS 第 2 个存储体

—

CS 第 3 个存储体

—

CS 第 4 个存储体

例如A

$$= 0, \quad A$$

$= 1$ ，则选择第 3 个存储体。

如果回忆一下第 16 章所讲的组织 RAM 阵列的繁琐细节，可能会认为还需要 8 个 4-1 选择器从 4 个存储体中选择正确的数据输入信号。但是，这并不需要。

通常，TTL 兼容集成电路的输出信号或者大于 2.2 伏（逻辑 1）或者小于 0.4 伏（逻辑 0）。

如果试着把输出信号连接起来，会出现什么情况呢？如果一个集成电路的输出为 1，另一个输出为 0，把这两个输出连接到一起后，结果会是什么呢？你是无法回答的。那么为什么集成电路的输出信号不能连接在一起呢？

2102 芯片的数据输出信号是三态输出。除了逻辑 0 和逻辑 1 外，数据输出信号还可以是第

三种状态，这种状态就是什么都不是，就好像芯片的引脚什么都没有连接。当 — 1 的

CS 输入为

时候，2102 芯片就进入第三种状态，这意味着可以把 4 个存储体对应的数据输出信号连接到一起，并用这 8 个组合输出作为总线的 8 个数据输入信号。

需要强调一下三态输出的概念，因为它对总线的操作至关重要。连到总线上的所有芯片

都可以使用总线上的数据输入信号向处理器传送数据。任何时候，连到总线上的众多电路板中只有一个用来确定总线上的数据输入信号是什么。其他电路板不被选中，输出为第三态。

2102 芯片是一个静态随机访问存储芯片，即 SRAM，与动态 RAM（DRAM）不同。

SRAM 每存储 1 位需要 4 个晶体管（与第 16 章讲到的用触发器作存储器所需要的晶体管数不完全相同），而 DRAM 每位只需要一个

晶体管。DRAM的缺点是它的外围支持电路较复杂。

只要芯片有电，SRAM芯片如2102就会保持已存储的内容；如果断电，则内容会丢失。在这方面，DRAM也是如此。但DRAM还需要周期性地对存储器进行访问，即使这些内容是不需要的。这称之为刷新，1秒钟内含有好几百次刷新，就好像隔一段时间就推一下某个人使他不要入睡一样。

抛开在使用DRAM上的争论，过去几年DRAM芯片容量的不断增长使得DRAM得到广泛

的应用。1975年，Intel公司首创了DRAM芯片，它可以存储16384位。与摩尔法则一致，DRAM芯片的容量基本上是每三年增长4倍。现在的计算机通常在系统板上有存储器槽，可以插上若干个DRAM芯片组成称为单行直插存储体（SIMM）或双行直插存储体（DIMM）的小电路板。现在128MB的DIMM售价在\$300以下。

既然已经知道了如何组织存储器，因此可不必把微处理器的存储空间都分配给存储器，

可以留一部分存储空间给输出设备。

阴极射线管（CRT）——外观上有些像在家里看到的电视机——已成为计算机最普通的输

出设备。连到计算机上的 CRT 通常称为视频显示器或监视器，提供信号给显示器的电子部件称为视频适配器。视频适配器是计算机中的一块电路板，通常称为视频卡。

虽然视频显示器或电视机的二维图像看起来似乎很复杂，但它实际上是由一束连续的射

线很快扫过屏幕而形成的。射线从左上角开始，从左到右扫过屏幕，然后很快回到左边，开始第 2 行。每个水平行称为扫描行，每次回到下一行的开始称为水平回扫。当射线扫描完最后一行后，就从屏幕右下角回到左上角（垂直回扫），并不断重复这一过程。以美国的电视信号为例，这种扫描每秒进行 60 次，称之为场频。由于扫描速度很快，图像在屏幕不会出现闪烁。

由于采用隔行扫描，电视的情况有点儿复杂。两个场需要用来形成一个单独的帧，帧是一个完整的静态视频图像。每个场完成整个帧的一半扫描线——第一个场完成偶数扫描线，第二个场完成奇数扫描线。水平扫描频率，即扫描每个水平行的频率，为 15 750 赫兹。把它除以 60 赫兹，为 262.5 行，这就是一个场的扫描线数。整个帧是它的两倍，即 525 条扫描线。

不考虑隔行扫描技术的细节，生成视频图像的连续射线由一个连续信号来控制的。尽管电视节目在进行广播或通过有线电视系统传送的时候是音频和视频的混和，但最终还是分开的。这里讲到的视频信号与从 VCR、摄像机和电视机的视频插口上输入/输出的信号是一致的。

对黑白电视来说，视频信号很简单也容易理解（彩色电视则要麻烦一些）。每秒 60 次场扫描，同时扫描信号中包含有用来标明一个场开始的垂直同步脉冲，脉冲电压为 0 伏（地），宽度为 400 毫秒。水平同步脉冲用来标明每个扫描行的开始，它的信号为 0 伏，宽度为 5 毫秒，每秒出现 15 750 次。在两次水平同步脉冲之

间，信号从 0.5 伏（黑）～ 2 伏（白）变化，0.5 伏～2 伏之间的电压用来表示灰度。

因而电视图像部分是数字的，部分是模拟的。图像在垂直方向上分成 525 行，但每一个扫描线的电压是连续变化的——用来模拟图像的可视强度。但是电压并不是无限制地变化，电视机能响应的信号变化频率有上限，称为电视机带宽。

带宽是通信中很重要的概念，它关系到某个传输媒体上能够传输的信息量。在电视机中，带宽限制了视频信号变化的速率。美国的广播电视带宽为 4.2MHz。

如果把视频显示器连接到计算机，则很难把显示器想像成是模拟和数字混合的设备，它很容易看成是一个地地道道的数字设备。从计算机的观点来看，视频图像很容易想像成是由离散的点组成的矩形网格，这些离散的点称之为像素。

视频带宽限制了水平扫描行上像素的个数。我们定义带宽为视频信号变化（从黑到白，再从白到黑）的速率。具有 4.2MHz 带宽的电视机允许每秒 420 万次 2 个像素或者——用水平扫描频率 15 750 去除 $2 \times 4\,200\,000$ ——每个水平扫描行 533 个像素。但是大约 1/3 的像素是不可用的，因为它们被隐藏了——或者在图像的远端或者在射线水平回扫中。水平扫描行上剩下大约有 320 个像素是有用的。

同样，在垂直方向上也不是 525 个像素都有用。实际上，在屏幕的上、下部和在垂直回扫过程中都会有损失。计算机在用电视机显示时不采用隔行扫描，垂直方向上的像素数目是 200。

因此，最初连到普通电视机上的视频适配器的分辨率为 320×200 ，即水平方向 320 个像素，垂直方向 200 个像素：

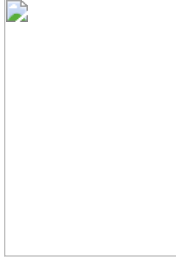


为了确定网格上像素总数，可以去统计也可以简单地用 320×200 得到 64 000 个像素。根

据视频适配器的配置（下面将会做简要的解释），每个像素可以是黑或白的，也可以为某一特定的颜色。

如果需要显示一些字符，能显示多少个呢？这显然依赖于每个字符的像素数目。下图是一种可能的方法，每个字符使用 8×8 网格

（64像素）：



这是ASCII码20h~7Fh对应的字符。（不可显示的字符对应的 ASCII码从00h~1Fh。） 每个字符由 7 位ASCII码来标识，而每个字符的显示情况由相对应的 64位来确定。可以把

这64位信息看成代码。

使用这种字符定义，可以在 320×200视频显示器中显示 25行，每行 40个字符。这足以显示Amy Lowell完整的一首短诗：



视频适配器需要一些 **RAM**来存储所显示的信息，微处理器可以向这个 **RAM**写入信息用来

改变屏幕显示的信息。更为方便的是，这个 **RAM**可以是微处理器存储空间的一部分。上面讲到的显示适配器需要多少 **RAM**呢？

这并不是一个简单的问题，结果可能的范围是从 **1KB~192KB**。从最低估计开始，减少视频适配器所需存储器的方法之一就是限制适配器只显示字符。

我们已知一屏可以显示 25行，每行 40个字符，即总共 1000个字符。这样视频适配器上的 **RAM**

只需存储这 1000个字符的 7位**ASCII**码。这1000个7位值大约是 1024字节即**1KB**。视频适配器还需要有包含所有 **ASCII**码字符点阵的字符生成器，这些字符生成器通常是只

读存储器，即 ROM。ROM是一种集成电路，特定的地址下得到特定的数据输出。不像 RAM， ROM没有数据输入信号。

也可把 ROM看成是把一种代码转换成另一种代码的转换电路。存储 128个ASCII码字符的 8×8点阵的 ROM需要7个地址信号（代表 ASCII码）和 64个数据输出信号。这样， ROM就把7 位ASCII码转换成可确定字符显示结果的 64位代码。然而 64位的输出信号会使得芯片过大！如 果有 10个地址信号、 8个输出信号就会变得很方便，其中 7位地址信号确定某一 ASCII码字符

（这 7位地址信号来自于视频适配器的数据输出）， 其余 3个地址信号用来标识字符中的一行。 如：地址 000标识最高行； 111标识最低行。 8位输出是每一行的 8个像素 。

例如，假设 ASCII码为41h。这是大写字母 A， 且有 8行， 每行 8位。下表显示了字符 A的10

位地址（空格分开了 ASCII码和行代码）和数据输出信号：

地址	数据输出
1000001 000	00110000
1000001 001	01111000
1000001 010	11001100
1000001 011	11001100
1000001 100	11111100
1000001 101	11001100
1000001 110	11001100
1000001 111	00000000

从中你是否看见了以 0 构成的背景上显示 1 来画出的 A? 显示字符的视频显示适配器必须还要有一个光标逻辑电路。光标是一个小下划线用来表

明键盘中输入的下一个字符将要显示的位置。光标字符的行, 列位置值通常存在视频卡的两个 8 位寄存器中, 微处理器可向其中写入数据。

如果显示适配器并非只限于显示文本, 则称为图像适配器。微处理器通过写入信息到图像视频卡上的 RAM 而在屏幕上显示图像, 其中包括各种大小和模式的字符。图像视频卡比仅显示字符的视频卡所需的存储器容量要大。显示 320×200 像素的图像视频卡有 64 000 个像素。如果一个像素对应于 1 位 RAM, 则该视频卡需要 64 000 位 RAM, 即 8000 字节。但是, 这是最低的要求。1 位对应于 1 个像素只能表示两种颜色——如, 黑和白。0 可能对应于黑色像素, 1 可能对应于白色像素。当然, 黑白电视机显示的不仅仅是黑、白两种颜色, 还可以显示出不同的灰度。为了在图像视频卡上显示灰度, 通常一个像素对应于一个字节的 RAM。00h 对应于黑色, FFh 对应于白色, 中间的数据对应于不同的灰度。显示 256 种不同灰度的 320×200 视频卡需要 64 000 字节的 RAM, 与一直在讲的 8 位微处理器的整个地址空间非常接近。

如果要达到很好的色彩效果, 则每个像素需要 3 个字节。仔细观察彩色电视机或计算机的视频显示器, 可以看到每种颜色是三原色, 即红、绿和蓝的不同混合。为了获得各种颜色, 需要一个字节来标明三原色每种颜色的强度。这样需要 192 000 字节的 RAM。(本书最后一章将会讲到更多有关彩色图形的内容。)

视频适配器能够显示的颜色多少与每个像素所用的位数有关。这种关系与本书中讲到的许多编码很相似, 也牵涉到 2 的幂:

$$\text{颜色数} = 2^{\text{每个像素使用的位数}}$$

320×200分辨率是标准电视机所能达到的最大分辨率，正因为如此，为计算机特制的显示器比电视机具有更高的带宽。1981年IBM PC所用的显示器可以显示25行，每行80个字符，这是IBM巨大且昂贵的大型机的CRT显示器上的字符数目。对IBM来说，80是一个很特殊的数字，因为它正好是IBM穿孔卡片上的字符数。的确，早期连到主机上的CRT显示器主要用来显示穿孔卡片的内容。偶尔你会听到一种过时的叫法，把仅显示字符的视频显示器的所有行称为卡片。

多年来，视频显示适配器的分辨率及显示的颜色不断增加。一个重要的里程碑是1987年IBM PS/2个人计算机系列和苹果公司的Macintosh II都采用了水平640像素，垂直480像素的视频适配器。这是从那时起就已保持的最低标准的视频分辨率了。

640×480分辨率是一个具有重要意义的里程碑。也许你不会相信，它之所以重要的原因

还要追溯到托尔斯·爱迪生！大约在1889年，当爱迪生和工程师William Kennedy Laurie Dickson研究电影摄影机和电影放影机的时候，他们决定使电影画面的宽比高要多出1/3。宽和高的比例称为长宽比。Edison和Dickson确定的这个比例通常表示为1.33:1或4:3；在60多年的时间里，这个长宽比为许多电影所采用，并且电视上也采用了这个长宽比。直到1950年的早期，好莱坞引入宽银幕技术来与电视竞争才打破了4:3的长宽比。

像电视一样，许多计算机监视器的长宽比也是4:3，这很容易用尺子测量一下来验证。

640×480分辨率的比例也是4:3，这意味着水平方向上100个像素的物理长度与垂直方向上

100个像素的物理长度是一样的。这是计算机图像的重要特征，称之为正方形像素。现在的视频适配器和监视器都能实现640×480的分辨率，但也存在其他各种各样的显示

模式，常见的分辨率有 800×600，1024×768，1280×960和 1600×1200。尽管总有人认为计算机显示器和键盘是按照同样的方式连到计算机上的——敲入什么就在

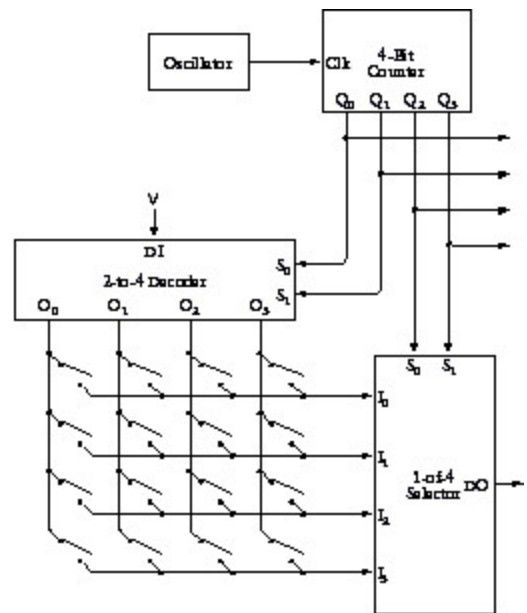
屏幕上显示什么——实际上，它们是不同的。键盘上的每一个键是一个简单的开关，键按下则开关闭合。可能有类似于打字机的 48 个

键的键盘，现在个人计算机键盘通常有 100 多个键。连到计算机上的键盘应该有硬件来为每一个按键提供唯一的代码，一种可能的方法是这

个代码是该键对应的 ASCII 码。但是，这种方法既不实用也不可取。例如，A 这个键可以对应于 ASCII 码 41h 或 61h，这取决于是否同时按下了 shift 键。此外，现在的计算机键盘有许多键并不对应于 ASCII 码字符，键盘硬件产生的代码是一种称之为扫描码的代码。一个小的计算机程序可以计算出在键盘上按下的某一个键时所对应的 ASCII 码（如果有的话）。

为使键盘硬件的描述不至于太繁杂，假设键盘只有 16 个键。无论什么时候按下一键，键盘硬件会产生一个 4 位代码，范围是 0000 ~ 1111。

键盘硬件包括了前面讲到的一些部件：



振荡器

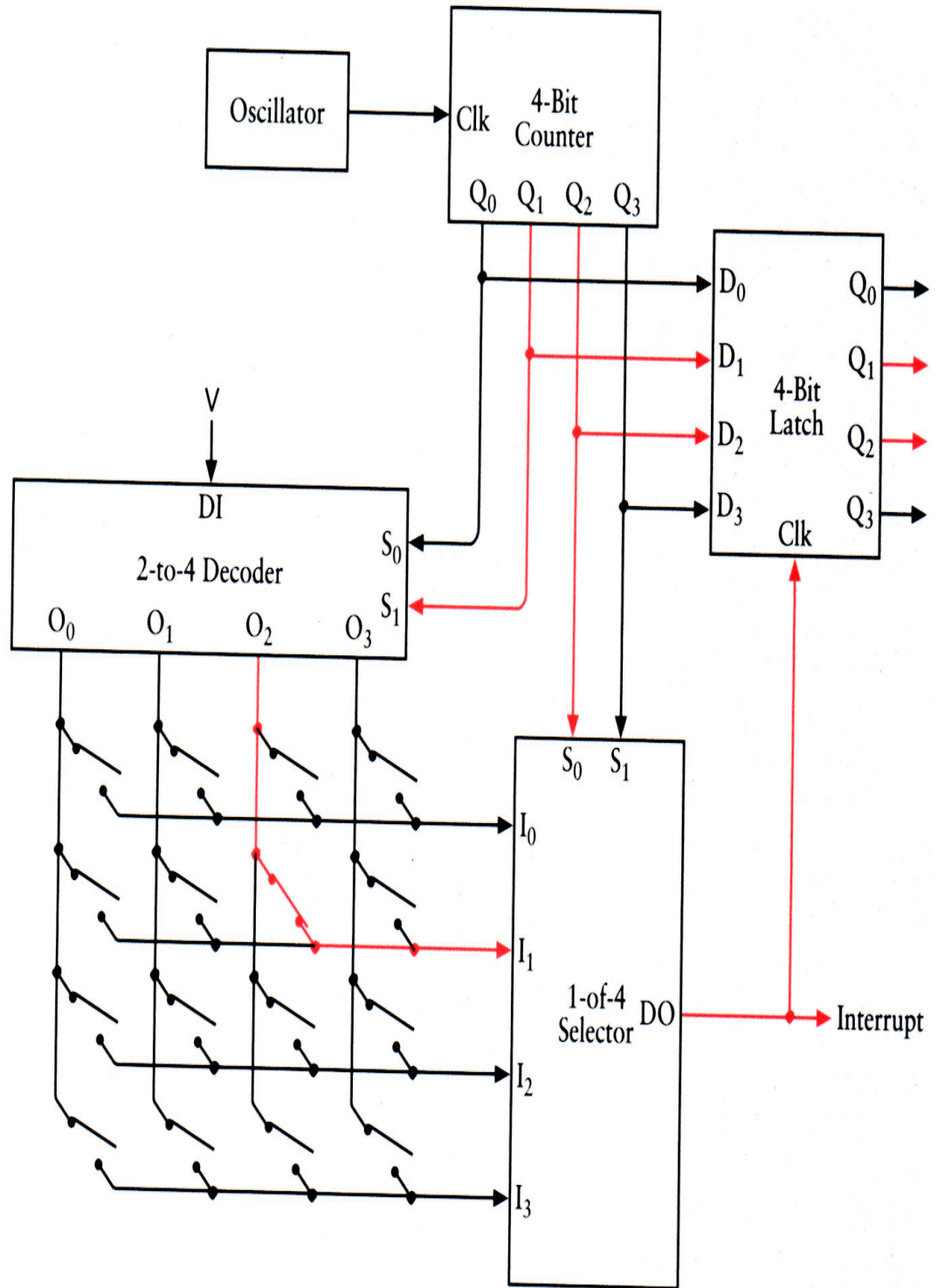
4位计数器

2-4 译码器

图的左下部是键盘的 16 个键，用简单的开关来表示。一个 4 位计数器迅速地循环对应于 16

个键的代码。循环所有的代码所需的时间必须比按下并松开一个键所需的时间要短。

4 位计数器的输出作为 2-4 译码器和 4-1 选择器的输入。如果没有键按下，选择器的输入没有一个为 1，则选择器的输出不为 1。如果有一个键按下，则对某一特定的 4 位计数器的输出来说，选择器的输出为 1。例如，如果从右上角开始的第二个键按下，并且如果计数器的输出为 0110，则选择器的输出为 1：



4位锁 存器

2-4 译码器

4-1 选

中断 译器

那就是该键所对应的代码。当该键被按下，没有其他计数器的输出将会使选择器输出为 1。每个键都有自己的代码。

如果键盘有 64 个键，则需要 6 位扫描码，也即需要 6 位计数器。可以用 3-8 译码器和 8-1 选择

器把键排成 8×8 阵列。如果键盘的键在 65~128 个之间，则需要 7 位代码。可以把键排成 8×16

阵列，采用 4-16 译码器和 8-1 选择器（也可用 3-8 译码器和 16-1 选择器）。下面将发生什么取决于键盘接口电路。键盘硬件可以为每一个键安排一个 1 位的 RAM，由

计数器的输出作为地址。如果键未按下，则 RAM 内容为 0；按下则为 1。可以由微处理器来读取 RAM 的内容以确定每个键的状况。

键盘接口中另一个有用的部分是中断信号。前面讲过，8080 微处理器允许外设中断 CPU 的当前工作。微处理器响应中断并从存储器中读入一条指令。这通常是一条 RST 指令，使微处理器转去执行内存中另外一个区域中的中断处理程序。

本章最后介绍的外围设备是长期存储设备。前面讲过，随机访问存储器——不论是用继电器、电子管还是晶体管构成的——在当电源关闭时，内容会丢失。所以，一个完整的计算机

也需要长期存储器。长久以来使用的方法是在纸上或卡片上打孔，就像 IBM 的穿孔卡片。在早期小型计算机中，是通过在滚动的纸带上打孔来保存程序 and 数据的这便于以后重新装入到内存中。但是，穿孔卡片和纸带存在一个问题，即介质不能重复使用，当打上一个孔后就很难再恢复。另一个问题是效率低，就当时来说，如果想要真切的看到某一比特，可能要花费太多空间。

所以，现在最流行的长期存储器类型是磁介质存储器。磁介质存储器起源于 1878 年，当时美国工程师 Oberlin Smith(1840—1926) 描述了它的原理。第一个可用的设备是在 20 年后即 1898 年，由丹麦的发明家 Valdemar Poulsen（1869—1942）制造的。Poulsen 的电磁式录音机起初打算用来记录人不在家时收到的电话信息。他用电磁铁—电报机里随处可见的部件—和可变长度的金属丝来记录声音。电磁铁按照声音的高低来磁化金属丝。当磁化的金属丝在通过电磁线圈的时候，根据磁化程度的不同会产生不同的电流。不管采用何种磁化介质，都是用电磁铁来记录和读取信息的。

1928 年，澳大利亚发明家 Fritz Pfleumer 发明了一种磁记录设备，该设备是在很长的纸带上采用最初用于生产香烟上金属带的技术覆盖铁粒子，并对它申请了专利。很快，一种强度更高的醋酸纤维素代替了纸，从而导致更耐久和更知名的记录介质的诞生。卷在轴上的磁带—现在都很方便地包装在塑料盒里—仍然是用来记录和回放音乐及视频信号的极通用的介质。

用来记录计算机数字数据的第一个商用磁带系统由 Reming Rand 在 1950 年发明。那时，1/2 英寸的卷轴磁带可以存放几兆字节的数据。早期家用计算机采用普通的盒式磁带录音机来保存信息。一些小程序用来存储内存块的内容到磁带并以后从磁带读到内存。最早的 IBM PC 有一个连接盒式磁带存储器的接头。今天，磁带仍然是很普遍的介质，特别是对那些要长期保存的文档。然而，磁带并不是理想的介质，因为不能很快地移动到磁带上的任一点进行访问，频繁的前进和倒回要花费很多时间。

从几何观点上看，能够进行快速访问的介质是磁盘。磁盘围绕中心旋转，连到臂上的一个或多个磁头从磁盘外边向中间移动。磁盘上的任何区域都能够被快速访问。

在记录声音信息这一方面，磁盘确实比磁带产生得要早一些。而用来存储计算机数据的第一个磁盘驱动器是由 IBM 在 1956 年

发明的，此 R A M A C （ random access method of accounting and control）由50个盘片组成，直径 2英尺，可以存放 5M字节数据。

从那时起，磁盘越来越小而容量越来越大。磁盘通常分为软盘和硬盘。软盘是由覆盖磁 性物质的塑料片组成，外面是起保护作用的厚纸板或塑料包装（现在常用）。（塑料包装保护 磁盘不被弯折，因而虽然现在的磁盘与以前的软盘已经有很大区别，但仍然习惯称之为软盘。） 软盘必须插入软盘驱动器，这是连接到计算机上的一个部件，用来向软盘写或从软盘读取信 息。早期的软盘直径为 8英寸。早期的 IBM PC用5.25英寸的软盘，现在常用的是直径 3.5英寸 的软盘。软盘可以从软盘驱动器中取出来，用来在计算机之间传递数据。磁盘现在仍然是商 用软件中一个重要的分发媒体。

硬盘通常由多个金属磁盘组成，永久性地做在驱动器里。硬盘通常比软盘速度快，并可 存储更多的数据。但是，硬盘中的磁盘自身不能移动。

磁盘的表面分成很多同心圆，称为磁道，每个磁道又分成像圆饼切片一样的扇区，每一 个扇区存放一定数量的字节，通常为 512 字节。最早 IBM PC 上用的软盘只有一面，分成 40个 磁道，每个磁道 8个扇区，每个扇区可保存 512字节。这样，每一个软盘可存放 163 840个字节， 即160KB 。今天， PC兼容机常用的 3.5英寸软盘有两面，每面 80个磁道，每个磁道 18个扇区， 每个扇区可存放 512字节，这样总共可存放 1 474 560字节，即 1440KB 。最早的硬盘驱动器由 IBM PC/XT 在1983年使用，可存放 10MB的内 容。 1999年， 20吉字节的硬盘驱动器（可存放 200亿字节）售价都只在 \$ 400 以下。

软盘和硬盘通常与它们的电气接口一起工作，它们与微处理器之间也需要另外的接口。硬盘驱动器常用的标准接口包括 SCSI (small computer system interface, 小型计算机系统接口)、ESDI (enhanced small device interface, 增强的小型设备接口) 和 IDE (integrated device electronics, 集成设备电气接口)，所有这些接口均使用 DMA (直接内存访问) 来接管总线和在随机访问存储器和硬盘之间直接传送数据，且不需经过微处理器。每次传输的数量是磁盘扇区字节数 (通常是 512 字节) 的倍数。

许多家用计算机的初学者总听到关于兆字节和吉字节的技术谈论，这使得他们对半导体随机访问存储器与磁盘存储器有什么不同感到很困惑。近几年出现的分类规则也减少了人们对术语的困惑。

随机访问存储器与磁介质存储器之间的主要区别是：随机访问存储器是易失性的，而软盘或硬盘上的数据会一直保留，除非故意删除或写覆盖。此外，还有一个显著的不同只有在理解微处理器如何工作之后才能理解：当微处理器输出一个地址信号后，通常是寻址随机访问存储器，而不是磁介质存储器。

从磁盘取出数据到内存供微处理器访问需要额外的步骤，即需要微处理器执行一段小程序去访问磁盘驱动器，使磁盘驱动器把数据传输到内存。

随机访问存储器与磁介质存储器之间的差别有一个比喻：随机访问存储器就像桌上的东西，可以直接拿来使用；磁介质存储器就像一个文件柜，如果要用文件柜里的东西，需要站起来，走到文件柜前，找到需要的文件，并带回到桌面上。如果桌面上太拥挤，还需要把桌上的一些东西拿回到文件柜中去。

这个比喻很恰当，因为存在磁盘上的数据确实是以所谓的“文件”来存放的。存放文件、提取文件是操作系统这个很重要软件的职权范围。

第 22 章 操作系统

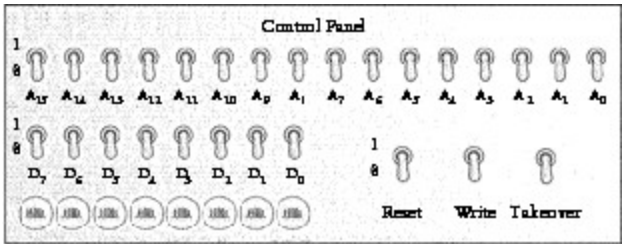
一直以来，我们似乎在组装着——至少在想像中——一台完整的计算机。它有一个微处理器、一些随机访问存储器、一个键盘、一个视频显示器和一个磁盘驱动器。当所有硬件各就各位以后，我们全神贯注于开关，给它加电，带给它生命。也许这样的描述会在你的脑海里唤起Victor Frankenstein 装配怪物时的情景，或者想起Geppetto正在制造将要命名为匹诺槽的木偶。

但我们还缺少一些东西，既不是惊人的力量，也不是良好的愿望。继续进行下去，打开新计算机电源，然后告诉我们你看到了什么？

当阴极射线管发热以后，屏幕上显示的是一些整齐排列但又是随机的ASCII码字符。这正如预期的那样，当电源断开时，半导体存储器的内容会丢失；当给它加电时，它处在随机的不可预料的状态。同样，为微处理器构建的所有RAM中的内容也是随机的，微处理器把这些随机的字节当作机器代码来执行。这样不会引起任何坏的情况发生，但是，也没有什么意义。

这里缺少的正是软件。当微处理器加电或复位时，它执行内存中某个地址里存放的机器代码。对8080来说，这个地址是0000h。对正确设计的计算机来说，加电时，该地址处应该有一个机器代码指令（很可能是多个指令中的第一条）。

机器代码指令又是怎样放到内存的那个地方的呢？在新设计的计算机中，把软件放到合适地方的处理过程可能是最令人费解的。要理解它，先从一个控制面板着手。该控制面板与第16章讲到的用来写入字节到随机访问存储器然后再读出的控制面板相似：



控制面板

复位 写入 接管

与以前的控制面板不同的是，这个控制面板有一个标明为复位的开关，这个开关连到微处理器的复位输入。只要这个开关是闭合的，处理器就什么也不做；当断开这个开关后，微处理器开始执行机器码。

控制面板的使用方法是：复位开关置 ON，复位微处理器，中止执行机器码；接管开关置

ON, 则接收总线上的地址信号和数据信号。这时, 可以使用 A
 $\sim A$

开关输入 16位的存储器地

址。标为 $D \sim D$ 的灯用来显示该地址的 8 位内容。要写入一个新的字节到相应的地址，则应在

15

0 7

$D \sim D$ 开关上设置该字节，然后把写入开关先拨到 ON 再拨到 OFF。完成了向内存中写入相应

0 7

字节以后，把接管开关设置为 OFF，复位开关设置为 OFF，则微处理器开始执行程序。这就是如何向刚刚从头建成的计算机中输入第一个机器码程序的过程，不用说，这是很

费事的。

又是什么改变了这一切，使得人们乐于在视频显示器前查看自己程序的执行结果呢？在上一章中已经讲到，只显示字符的视频显示器有 1KB 的随机访问存储器用来存放 25 行，每行 40 个字符的 ASCII 码。程序把内容写入到该存储器中，方法与写入到计算机中其他存储器中的方法一样。

然而，把程序的输出显示到视频显示器并不是那么简单。例如，如果一段程序，执行结果是 4Bh，则不能简单地把这个值写入视频显示器的存储器中。如果这样做，屏幕上将会看到的是字符 K，因为该字符对应的 ASCII 码是 4Bh。正确的是应写两个 ASCII 码字符到显示器：34h（是 4 的 ASCII 码）和 42h（是 B 的 ASCII 码）。8 位的计算结果每半个字节是一个十六进制数字，该数字必须通过对应的 ASCII 码来显示。

当然，也可以写一段小的子程序来完成这种转换。下面的一段 8080 汇编语言程序用来把十六进制数中的一位转换成对应的 ASCII 码（假定包含的十六进制数范围从 00h~0Fh）：

```
NibbleToAscii: CMP A,0Ah ;Check if it's a letter or number (判断是数字还是字母)

                JC Number

                ADD A,37h          ;A to F converted to 41h to 46h (把 A ~ F 转换成 41h ~ 46h) RET

Number:         ADD A,30h          ; 0 to 9 converted to 30h to 39h( 把 0 ~ 9 转换成 30h ~ 39h) RET
```

下面的子程序调用 NibbleToAscii 两次，把累加器 A 中的一个字节转换成两个 ASCII 码数字，并放在寄存器 B 和 C 中：

```

ByteToAscii:      PUSH                PSW                                ;Save accumulator( 保存 A)

                  RRC RRC RRC RRC                                ;Rotate A right 4 times... ( A右移 4次 ...)

                  ;...to get high-order nibble( 取高半字节 )

                  CALL NibbleToAscii;Convert to ASCII code( 转换成 ASCII 码 )

                  ;入寄存器 B)

MOV               B,A                                ;Move result to register B (结果放

POP AND          PSW A,0Fh                                ;Get original A back (取出原来的 A)

                  ;Get low-order nibble (取低半字节

)

CALL NibbleToAscii      ;Convert to ASCII code (转换成 ASCII 码) MOV C,A      ;Move result to
register C (结果放入寄存器 C) RET

```

这些子程序使得可以在视频显示器中按十六进制来显示一个字节。如果要转换成十进制，再做一些工作即可。此过程与把十六进制数转换成十进制数的方法非常相似——用10来除几次即可。

记住，还没有把这些汇编语言程序输入到内存中。也许，你已经把它们写到了纸上并且转换成了机器码，然后再输入到内存中。这种“手工汇编”是第24章要讲的内容。

尽管控制面板不需要许多硬件，但却不容易使用。它所采用的输入/输出方法是最坏的方法。既然聪明到可以从零开始来制造自己的计算机，却还用数字 0 和 1 来作为按键，的确令人

汗颜。那么首先要做的是去掉控制面板。当然要用键盘来作为按键。前面讲过计算机键盘的构造是只要按下一个键，就会产生一

个对微处理器的中断信号。计算机中的中断控制芯片使得微处理器响应中断，执行一条 **RST** 指令。假设这是一条 **RST 1**指令，这条指令使得微处理器在堆栈中保存当前程序计数器的值并 跳转到地址 **0008h**处。从这个地址开始，可以输入一些代码（用控制面板）。这些代码称为键 盘处理程序。

为了使一切都正常工作，还需要一些代码在微处理器复位时执行，这些代码叫 初始化程序。初始化程序首先设置堆栈指针，使得堆栈分配到内存的有效区域，然后，把视频显示存 储器的每一个字节设置为十六进制数 **20h**，即ASCII码的空格，这样就可以去掉屏幕上的随机 字符。初始化程序用 **OUT (Output)** 指令设置光标的位置（光标是视频显示器上的下划线，指示了新输入的字符将要显示的位置）到第 1行第1列。下一条指令为 **EI**，即中断允许，该指 令使得微处理器可以响应键盘中断。在此之后是 **HLT**指令，它停止微处理器的工作。

这就是初始化程序的工作。从这时起，由于执行了 **HLT**指令，计算机很可能处于停机状 态。能够把计算机从停机状态唤起的事件仅有来自于控制面板的复位信号或从键盘来的中断 信号。

无论何时在键盘上按下一个键，中断信号都使得微处理器从初始化程序最后的 **HLT**语句 跳转到键盘处理程序。键盘处理程序用 **IN (Input)** 指令来确定按下的键，然后根据按下的键 来执行一些动作（即键盘处理程序处理每一个按键），接着执行一条 **RET (Return)** 指令，最后又回到 **HLT**语句等待另一个键盘中断。

不论按下的是字符、数字还是标点符号，键盘处理程序使用键盘扫描码，结合 **Shift**键是 否被按下，来确定合适的 **ASCII**码。然后将 **ASCII**码写到视频显示存储器中光标的位置。这个 过程称为回

显键到显示器。光标位置增加并移到刚才显示的字符后面的空格处。由此，可以在键盘上敲入一串字符并显示在屏幕上。

如果按下的键是 **Backspace**（对应的 **ASCII**码是08h），则键盘处理程序删除最后写入到视频显示存储器中的字符，（删除字符是很简单的一件事，只需写入 **ASCII**码 20h—空格字符

—到某一内存位置。）然后把光标移回一格。人们通常在键盘上敲入一行字符（需要改正错误时可用 **Backspace** 键），然后敲入

Return(回车)键，回车键在计算机键盘上通常标为 **Enter**。与在电子打字机上敲 **Return**键表明已经准备好开始输入下一行一样，在计算机中敲 **Enter**键表明打字者已经完成了一行文字的键入。

键盘处理程序在处理 **Return**或**Enter**键（对应的 **ASCII**码为0Dh）的时候，视频显示存储器的这一行字符被解释成对计算机的一个命令，也就是说，键盘处理程序要去做的一些事情。键盘处理程序中包含有命令处理程序用来解释命令，例如三个命令：**W**、**D**和**R**。

如果字符行以 **W**开始，该命令意味着 **Write**（写入）一些字节到内存中。假设敲入到屏幕上的行如下面这样：

W 1020 35 4F 78 23 9B AC 67

这个命令指示命令处理程序把十六进制数 35、4F等写入到地址 1020h开始的内存中。为了完成这项工作，键盘处理程序需要将 **ASCII**码转换成字节—前面示范的那个变换的反变换。

如果字符行以 **D**开头，该命令意味着 **Display**（显示）内存中的一些字节。假使敲入到屏

幕上的行如下面这样：

D 1030

命令处理程序将会显示从内存地址 1030h开始的存放在内存中的 11个字节（之所以为 11，是因为在 40个字符宽的显示器上，在与上面命令同一行的地址后面能显示的字符数为 11）。可以用 Display命令来查看内存中的内容。

如果字符行以 R开头，该命令意味着 Run（运行），如下的命令：

R 1000

意味着“运行从地址 1000h处开始存储的程序”。命令处理程序把 1000h存到寄存器对 HL 中，然后执行指令 PCHL，即把寄存器对 HL的值装入程序计数器，也就是跳转到该地址处执行程序。

采用键盘处理程序和命令处理程序进行工作是一个重要的里程碑。有了它，无需再用什么控制面板，从键盘输入容易、迅速且效果良好。

当然，还有问题。当电源断电时，输入的所有代码会丢失。正因为如此，可能要把这些新代码存到只读存储器，即 ROM中。上一章曾讲到了一个 ROM芯片里存有所有用来在屏幕上

显示ASCII字符的点阵模式。假定所用的芯片在制造时已经配置有这些数据，则你也可以在家里自己编程 ROM芯片。可编程只读存储器（PROM）芯片只可以编程一次；可擦除可编程只读

存储器（EPROM）芯片即可以编程，而且它在紫外光的照射下擦除所有的信息后还可以重新再进行编程。

前面讲过，RAM板连到 DIP开关，DIP开关允许设定 RAM板的开始地址。如果使用的是

8080 系统，初始时一个 RAM 板地址应设置成 0000h。如果还有 ROM，则 ROM 的地址应为

0000h，而 RAM 板可以连到更高的地址。命令处理程序的创建是一个重要的里程碑，不仅因为它对输入到内存中的字节提供了较快的解释，而且使计算机现在成为交互式的了。当从键盘上敲入一些东西后，计算机就会做出响应，并在屏幕上显示出来。

一旦有了 ROM 中的命令处理程序，就可以开始试着从内存中写入数据到磁盘驱动器（可能是对应于磁盘扇区大小的块），并且把数据读回到内存。把程序和数据存放在磁盘上比存放

在 RAM 中要安全得多（后者如果电源出故障它们会丢失），也比存放在 ROM 中要灵活得多。也许应该加入一些命令到命令处理程序，如用 S 命令来表示存储：

S 2080 2 15 3

这个命令表示从地址 2080h 处开始的内存块将要存放到磁盘的第 2 面，第 15 磁道，第 3 扇区

（内存块的大小根据磁盘扇区的大小确定）。同样，也可以加入一个 Load 命令：

L 2080 2 15 3

该命令把该扇区的内容从磁盘送回到内存中。当然，还需要保留存放的地方的记录，可以用手边的本和铅笔来记录。一定要小心不要

把保存在某个地址的代码重载到内存的另一个地址，这样做就别指望它能正常工作。所有的

Jump 和 Call 指令将会出错，因为它们标识的是原来的地址。同样，如果一个程序比磁盘扇区的大小要大，则需要把它存放到几个扇区。磁盘中有些扇区可能被其他程序或数据占用了，有些扇区还是空的，因而存放长程序的扇区在磁盘上可能是不连续的。

这样，你可能就会发现手工记录哪些东西存放 to 哪些地方的工作是相当多的，正因为如

此，就需要有一个文件系统。文件系统是指在磁盘存储器中按文件来组织数据的方法。文件是存放在一个或多个扇区

中相关数据的集合。更重要的是，每个文件有一个文件名作为标识，便于记住文件中包含的内容。可以把磁盘看成类似于文件柜，里面的每一个文件都有一个标志用来表示文件的名称。文件系统通常是称作操作系统的较大软件集合的一部分。本章构造的键盘处理程序和命令处理程序也肯定包含在操作系统中。先不考虑其漫长的演化过程，让我们看一下真正的操

作系统是在干什么，又是如何工作的。

回顾历史，最重要的 8 位微处理器操作系统是 CP/M，是 Gary Kildall（出生于 1942 年）在 20 世纪 70 年代中期为 Intel 8080 微处理器而写的，他后来创立了 D R I（digital research incorporated）公司。

CP/M 存放在磁盘中。早期 CP/M 最常用的存储介质是单面 8 英寸磁盘，有 77 个磁道，每道 26 个扇区，每扇区 128 个字节（总共 256 256 字节），磁盘的头两个磁道包含有 CP/M。下面将简单地描述 CP/M 是如何从磁盘装入到计算机内存中的。

CP/M 盘中余下的 75 个磁道用来存储文件。CP/M 的文件系统虽然很简单，但却满足两个基本的要求：首先，磁盘中的每个文件有一个名字作为标识，这个名字也存在磁盘中。其实，CP/M 用来读取文件所需的全部信息都与文件一起存放在磁盘中；第二，文件在磁盘中并不占据连续的扇区。由于经常创建和删除不同大小的文件，因而磁盘上的剩余空间都是碎片。文件系统具有把大文件存放在不连续扇区的这种能力是非常有用的。

用来存放文件的 75 个磁道按分配块进行分组，每一个分配块有 8 个扇区，即 1024 字节。磁盘中共有 243 个分配块，编号从 0～242。

开始的两个分配块（共 2048字节）用作目录区。目录区是磁盘中的一个特殊区域，用来 存放磁盘中每一个文件的名称和一些主要信息。存在磁盘中的每一个文件需要一个 32字节长的目录项。因为目录区总共只有 2048字节，因而磁盘能够存放 2048/32，即64个文件。

每一个32字节的目录项包含有以下信息：

字节	含义
0	通常设为 0
1 ~ 8	文件名
9 ~ 11	文件类型
12	文件扩展
13 ~ 14	保留（设置为 0 ）
15	最后一块的扇区数
16 ~ 31	磁盘存储表

目录项的第一个字节只在文件系统可供两个或更多人同时共享时使用。在 CP/M中，该字 节通常设置为 0，与第13、14字节一样。

在CP/M 中，每个文件的文件名由两部分组成，第一部分称作文件名，最多有 8个字符，

存放在目录项的第 1~8字节；第二部分是文件类型，最多有 3个字符，存放在第 9~11字节。有几个标准的文件类型，如： T X T 表示文本文件（即文件中只包含 A S C I I 码）， C O M

（Command的简称）表示文件内容是 8080机器码指令或程序。定义文件时，这两部分由点隔 开，如：

这种文件命令的方式习惯上称为 8.3，表明点前最多有 8 个字符，点后最多有 3 个字符。目录项中的磁盘存储表标明了该文件所存放的分配块。假设磁盘存储表的前 4 项分别为

14h、15h、07h 和 23h，其余均为 0，则表明该文件占用 4 个分配块，即 4KB 的空间。文件实际上可能要短一些。目录项的第 15 字节标明在最后一个分配块中实际用到了多少个 128 字节的扇区。

磁盘存储表长 16 字节，可以容纳长达 16 384 字节的文件，超过 16KB 的文件要使用多个目录项，称为扩展。在这种情况下，第一个目录项的第 12 字节设置为 0，第二个目录项的第 12 字节设置为 1，依此类推。

上面提到过文本文件也称为 ASCII 文件，或其他类似名称。文本文件中包含有对应于字符的 ASCII 码（包括回车和换行代码）供人们浏览。不是文本文件的文件称为二进制文件。CP/M 的 COM 文件为二进制文件，因而它包含 8080 的机器码。

假设一个文件（一个很小文件）包括三个 16 位数——例如，5A48h、78BFh 和 F510h。由这三个数字组成的二进制文件长仅为 6 字节：

```
48 5A BF 78 10 F5
```

当然，这是存储多字节数的 Intel 格式，其中低字节在前。为 Motorola 处理器编写的程序则是按以下方式来创建文件：

```
5A 48 78 BF F5 10
```

若用 ASCII 码文本文件存放这同样 3 个 16 位数，则由以下这些字节组成：

```
35 41 34 38 68 0D 0A 37 38 42 46 68 0D 0A 46 35 31 30 68 0D 0A
```

这些字节是数字和字符的 **ASCII**码，每一个数由回车（**0Dh**）和换行（**0Ah**）终止。文本文件很容易显示，它们不是作为字节串，而是作为字符显示：

5A48h

78BFh F510h

包含这3个数的**ASCII**码文本文件也可以由以下字节组成：

32 33 31 31 32 0D 0A 33 30 39 31 31 0D 0A 36 32 37 33 36 0D 0A

这些字节是与这 3个数等效的十进制数的 **ASCII**码：

23112

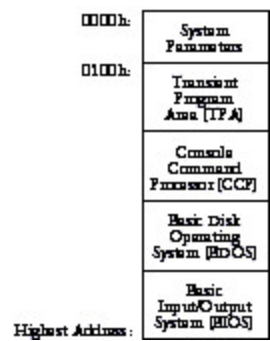
30911

62736

既然采用文本文件的目的是方便人们阅读，因而没有什么理由不用十进制而非要用十六进制。

上面提到过，**CP/M**自身存放在磁盘的头两个磁道。为了执行它，**CP/M**必须从磁盘装载到内存。使用**CP/M**的计算机中，**ROM**并不需要很多，它只需要用来存放一小段代码，称为引导程序（因为这段代码通过自举来引导操作系统的其余部分）。引导程序把磁盘最开始的 128 个字节的扇区装入内存并执行，这个扇区包含有把**CP/M**的其余部分装入内存的代码。整个这个过程称为引导操作系统。

最终， CP/M把它自己安排在 RAM的最高地址区域。装载 CP/M以后， 整个内存组织如下 所示：



系统参数

临时程序区域

(TPA)

控制台命令处理程序

(CCP)

基本磁盘操作系统

(BDOS)

最高地址

基本输入 / 输出系统

(BIOS)

该图不是按比例画的。CP/M的三个部件——基本输入/输出系统（BIOS）、基本磁盘操作系统（BDOS）和控制台命令处理程序（CCP）仅占用6KB的内存，临时程序区域（TPA）

——在64KB内存的计算机中大约有58KB——初始时没有任何东西。控制台命令处理程序等效于前面构造的命令处理程序，控制台指的是键盘和显示器。

CCP在显示器上显示提示符，就像这样：

```
A >
```

提示符提示可以输入信息。在有不止一个磁盘驱动器的计算机中，A指的是第一个磁盘驱动器，CP/M从该驱动器装入。在提示符后敲入命令并按回车键，CCP就执行该命令并在屏幕上显示结果信息。命令执行完以后，CCP又显示提示符。

CCP只能识别一些命令，最重要的命令可能是：

```
DIR
```

该命令用来显示磁盘目录，即存放在磁盘中的所有文件的列表。可以用特殊字符?和*来限定显示具有某些特定名称和类型的文件，例如：

```
DIR *.TXT
```

显示所有文本文件，而

```
DIR A???? B.*
```

显示文件名为5个字符，第一个字符为A，最后一个字符为B的所有文件。另外一个命令是ERA，它是Erase的缩写，用来从磁盘中删除文件。例如：

```
ERA MYLETTER.TXT
```

删除具有这个名字的文件，而：

ERA *.TXT

删除所有文本文件。删除文件意味着释放文件的目录项及文件所占用的磁盘空间。还有一个命令是 **REN**，它是 **Rename** 的缩写，用来改变文件名。 **TYPE** 命令用来显示文本文

件的内容。因为文本文件只包含有 **ASCII** 码，因而该命令还可用来浏览屏幕上的文件内容， 如：

TYPE MYLETTER.TXT

SAVE命令用来把临时程序区域中的一个或多个 256字节的内存块以一个特定名称存入到磁盘中。

如果敲入一个 **CP/M**不能识别的命令，就认为输入的是磁盘中的一个程序的名称。程序的文件类型为 **COM**，代表命令。**CCP**在磁盘中查找叫这个名字的文件，如果有，**CP/M**把文件从磁盘装入临时程序区域，该区域从地址 **0100h**处开始。以上就是告诉你如何运行磁盘中的文件。如果在**CP/M**提示符后敲入：

CALC

且如果名称为 **CALC.COM**的文件存在于磁盘中，则 **CCP**把该文件装入从地址 **0100h**处开始的内存中，然后转到地址 **0100h**处的机器码指令开始执行程序。

前面讲述了如何在内存的任一地方加入机器码指令并执行，但按磁盘文件存储的 **CP/M**程序必须设计成从内存的特定地址 **0100h**处开始装入。

CP/M包括几个有用的程序，如 **PIP**（**peripheral interchange program**），即外设交换程序，用来拷贝文件。**ED**是文本编辑器，用来创建和修改文本文件。像 **PIP**和**ED**这类小且用来完成简单事务的程序通常称为实用程序。如果运行 **CP/M**系统，可以购买一些大的应用程序，如字处理软件或计算机电子报表软件；也可以自己编制这样的软件。所有这些也都以 **COM**类型的文件存储。

到目前为止，已经知道了 **CP/M**（像许多操作系统一样）如何提供命令和实用程序以便对文件进行基本的操作。同样，也已经知道 **CP/M**如何把程序装载到内存并执行。作为一个操作系统，**CP/M**还有第三个主要功能。

在**CP/M**下运行的程序经常需要把输出写到视频显示器，或者从键盘上读入输入的内容，或者从磁盘读取一个文件和向磁盘中写入

一个文件。但通常情况下，**CP/M**程序并不把程序输出直接写到视频显示存储器中；同样，**CP/M**程序也不访问键盘硬件看看输入了什么，它也不访问磁盘驱动器硬件去读或写磁盘的扇区。

事实上，运行在**CP/M**下的程序利用**CP/M**中所构建的子程序集来完成这些公共事务。这些子程序经过特别设计，从而使得程序很容易访问计算机中的硬件——包括视频显示器、键盘和磁盘——且程序设计员不用关心这些外设实际上是怎样进行连接的。更重要的是，在**CP/M**下运行的程序不需要了解磁道、扇区，这是**CP/M**的工作，它可以把文件存放到磁盘，也可以读取磁盘上的文件。

为程序提供方便访问计算机硬件的手段是操作系统的第三个主要功能。操作系统提供的这种访问手段称之为应用程序接口，即**API**（**application programming interface**）。

在**CP/M**下运行的程序通过设置寄存器 **C** 为某一特定值（叫作功能值）来使用 **API** 并执行指

令：

```
CALL 5
```

例如，一个程序通过执行下面的指令获取从键盘上输入的键的 **ASCII** 码：

```
MOV C, 01h CALL 5
```

累加器 **A** 中包含有输入的键的 **ASCII** 码。同样

```
MOV C, 02h
```

CALL 5

把累加器 A 中的 ASCII 码字符写到视频显示器中光标的位置，光标移到下一个位置。如果程序中要创建一个文件，则把寄存器对 DE 设置为包含有文件名所在的内存区域的地

址，然后执行以下代码：

```
MOV C, 16h CALL 5
```

此例中，CALL5 指令使 CP/M 在磁盘上创建一个空文件。程序可以利用其他功能向文件写入，最后关闭文件，意味着文件已经使用完毕。该程序和其他程序以后可打开文件并读取文件内容。

CALL5 到底能做什么呢？在内存 0005h 位置由 CP/M 设置了一条 JMP (Jump) 指令，该指令跳转到 CP/M 基本磁盘操作系统 (BDOS) 所在的位置。这个区域包含有一些子程序用来完成 CP/M 的每一项功能。BDOS 正如它的名字一样，基本作用是维护磁盘上的文件系统。通常 BDOS 必须利用 CP/M 基本输入/输出系统 (BIOS) 中的子程序，而 BIOS 可实现对像键盘、视频显示器以及磁盘驱动器这样的硬件的访问。实际上，BIOS 是 CP/M 中唯一需要了解计算机硬件的部分。CCP 利用 BDOS 的功能来实现自己功能，那些 CP/M 提供的实用程序也是如此。

API 是与设备无关的计算机硬件接口，也就是说在 CP/M 下编写的程序不需要知道某一机器上键盘的工作机制、视频显示器的工作机制或读写磁盘扇区的工作机制，它只是简单地利用 CP/M 的功能来完成涉及到键盘、显示器和磁盘的工作。这样，CP/M 程序就可以在不同的

计算机上运行，而这些机器可能会用差别很大的硬件来访问外设。（所有 CP/M 程序必须运行

在8080微处理器上，或能执行8080指令的处理器上，如：Intel 8085或Zilog的Z-80。）只要计算机运行CP/M，则程序就可以利用CP/M的功能间接访问硬件。如果没有标准的API，程序则需要针对不同类型的计算机来做不同的工作。

CP/M曾经是8080中非常流行的操作系统，至今仍具有重要的历史意义。CP/M对其后的16位操作系统QDOS（quick and dirty operating system）有很大的影响。QDOS是西雅图计算机产品公司（seattle computer products）的Tim Paterson为Intel的16位8086和8088芯片而编写的。QDOS后来改名为86-DOS，由Microsoft公司注册。该操作系统被授权给IBM以MS-DOS这个名称用于第1代IBM PC机。尽管CP/M的16位版本（称为CP/M-86）也可用于IBM PC，但MS-DOS很快成了标准。MS-DOS（在IBM计算机上叫PC-DOS）也允许其他生产IBM PC兼容机的厂商使用。

MS-DOS没有保留CP/M的文件系统，在MS-DOS文件系统中使用的是一张叫文件分配表的表，即FAT。这种技术最初由Microsoft公司在1977年采用。磁盘空间分成簇，根据磁盘空间大小，簇的大小也从512~16384字节不等。每个文件是簇的集合，文件的目录项只表明了文件开始的簇，FAT能够表明磁盘上每一个簇的下一簇。

MS-DOS磁盘上的目录项长32字节，采用与CP/M一样的8.3文件命名系统，只是术语有些不同：后面的3个字符称作文件扩展名而不是文件类型。MS-DOS的目录项无需包含分配块的

列表，它包含的是这样一些有用的信息，如文件最后修改的日期、时间及文件大小。

MS-DOS的早期版本在结构上很像CP/M，但MS-DOS中不需要BIOS，因为IBM PC中已经有完整的BIOS存放在ROM中。MS-DOS的命令处理程序是一个名叫COMMAND.COM的文件。MS-DOS的运行程序有两种：具有扩展名COM的文件，大小不能超过64KB；具有扩展名EXE（可执行）的较大文件。

尽管开始时 **MS-DOS** 支持 **CALL 5 API** 功能接口，但对新的程序推荐了新的接口。新的接口利用了 **8086** 的一个功能叫作软件中断，这类似于子程序调用，但程序不需要知道它正在调用的确切地址。程序通过执行指令 **INT 21h** 调用 **MS-DOS** 的 **API** 功能。

理论上讲，应用程序只能通过操作系统提供的接口它们来访问计算机的硬件。但对针对 20 世纪 70 年代和 80 年代早期的小型操作系统的应用程序而言，经常绕过操作系统，尤其是在处理视频显示器的时候。直接写入字节到视频存储器的程序比采用其他方式的程序执行速度要快。的确，对有些应用程序——例如，那些需要在显示存储器上显示图形的应用程序——操作系统是不合适的。**MS-DOS** 最吸引程序员的地方正是它的“反传统性”，程序员可以编写程序以达到硬件的最快速度。

正因为如此，运行在 **IBM PC** 上的流行软件常常是根据 **IBM PC** 的硬件特点编制的。机器制造商为了与 **IBM PC** 竞争也不得不沿袭这些特点。如果不这样做，则会使得这些流行软件不能运行。这些软件通常要求硬件是“**IBM PC** 或与 **IBM PC** 100% 兼容”。

MS-DOS 2.0 版于 1983 年 3 月发布，它增强了功能来使用硬盘驱动器。虽说当时的硬盘容量很小（按今天的标准），但很快就变得大了起来。当然，硬盘越大就越能存储更多的文件，但磁盘上存储的文件越多，则找到某个文件或组织文件就变得越麻烦。

MS-DOS 2.0 的解决方法是采用层次文件系统，它对原有的 **MS-DOS** 文件系统做了一些小的改动。前面讲过，磁盘有一个区域叫目录，它是一个文件列表，里面包含了有关文件存放在磁盘的什么地方的信息。在层次文件系统里，一些这样的文件可能本身就是目录，也就是说，它们是包含其他文件列表的文件，这些文件也有可能还是目录。磁盘中，这个常规的目录称为根目录，包含在其他目录里的目录称为子目录。目录（有时称文件夹）成为对相关文件进行分组的一种方法。

层次文件系统以及MS-DOS 2.0的其他一些功能是从UNIX操作系统借鉴来的。UNIX是20世纪70年代早期在贝尔实验室开发的，大部分工作由 Ken Thompson（生于1943年）和 Dennis Ritchie（生于1941年）完成。这个操作系统有趣的名字是一个文字游戏：UNIX先是作为贝尔实验室为MIT和GE开发的名为 Multics（表示多路复用信息和计算业务：multiplexed information and computing services）的早期操作系统的一个缺少健壮性的版本。

对设计计算机核心程序的计算机程序员来说，UNIX什么时候都是很好的操作系统。虽然大多数操作系统都是针对特定计算机的，但UNIX是可移植的，意思是它可以运行在各种各样的计算机中。

在开发UNIX的时候，贝尔实验室还是AT&T的一个辅助机构。为了抑制AT&T在电话业的垄断地位，AT&T受到法庭裁决。起初，AT&T被禁止销售UNIX，公司被迫把它授权给别人。所以从1973年开始，UNIX被广泛授权给大学、公司和政府机构。1983年，AT&T获准重返计算机业并发布了它自己的UNIX版本。

由此导致的结果就是没有单一的UNIX版本，相反，有许多不同的版本，用不同的名称，运行在不同的计算机上并由不同的经销商销售。许多人把手伸向UNIX，并在UNIX上留下印迹。然而，当人们在UNIX上加一些东西时，似乎仍然有一种流行的“UNIX哲学”在引导人们。这个哲学的其中一部分是用文本文件作为公用的文件形式。许多UNIX实用程序读取文本文件，利用它们来做一些工作，然后写入另外一个文本文件。UNIX的实用程序可以组织起来形成一个链，然后在这些文本文件上实现不同的处理。

UNIX最初是为只一个人使用时大且昂贵的计算机而编写的。使用UNIX的计算机通过分时技术允许多个用户同时与计算机交互操作。由于很快地在所有终端之间切换时间片，UNIX操作系统使得用户感觉计算机就像在同时为每个人服务。

并行运行多道程序的操作系统称为多任务操作系统。显然，这种操作系统比像CP/M和MS-DOS这样的单任务操作系统要复杂得多。多任务使得文件系统复杂化，因为多个用户可能会试图同时访问同一个文件。多任务同样也影响到计算机如何为不同程序分配内存，所以需要进行内存管理。由于多道程序并行运行需要更多的内存，因而很可能计算机没有足够的内存来分配。操作系统可能需要采用虚拟内存技术，当程序不需要某些内存块时可以把它们存放在临时文件中，等需要时再读回内存。

近几年来，UNIX最令人感兴趣的发展是FSF（自由软件基金会，free software foundation）和GUN方案，它们都由Richard Stallman建立。GUN表示“GUN不是UNIX”，当然，GUN不是UNIX。GUN试图与UNIX兼容但却采用了一种方式来使得软件不成为专有的。GUN方案导致了許多与UNIX兼容的实用程序和工具，还有Linux，它是一个与UNIX兼容的操作系统的内核。Linux的大部分程序由芬兰的Linus Torvalds完成。近几年，Linux已经变得很流行。

从20世纪80年代中期开始，操作系统最显著的发展趋势是开发大型的、成熟的操作系统，如，苹果公司的Macintosh和微软的Windows，它们结合了图形和可视化视频显示，从而使其更容易使用。本书最后一章将要描述这种趋势。

第 23 章 定点数和浮点数

日常生活中，有各种各样的数，整数、分数、百分数等等，我们无时无刻不与这些数打交道。如：用加班 2.75 小时获得的 1 倍半的钱来买半筐鸡蛋需支付 8.25% 的销售税。许多人对诸如此类的数都感到很适应，并不需要怎么在行，即使在听到“平均每个美国家庭有 2.6 人”这样的统计数字的时候，也不会联想到 2.6 这个数字对人来说是不是要把人肢解了这样可怕的问题。

在计算机内存里，整数和分数的换算是常见的。存在计算机内存里的东西都是二进制位的形式，也就是说，都是二进制数。但有些数用位来表示比其他数用位来表示要容易一些。

我们使用位来表示数学上称为自然数而计算机编程人员称为正整型数的数，并介绍如何用 2 的补码来表示负整数，而这种方法很容易实现正数、负数的加法。下表列出了 8 位、16 位、32 位的正整数及它们的 2 的补码的范围：

数的位数

正整数范围

2 的补码范围

8

0 ~ 255

- 128 ~ 127

16

0 ~ 65 535

-32 768 ~ 32 767

32

0 ~ 4 294
967 295-2 147 483 648 ~ 2 147
483 647

要介绍的就是这些。除了整数以外，数学上还定义了有理数，它们可表示成两个整数的比，这个比也叫分数。例如， $\frac{3}{4}$ 是一个有理数，因为它是3与4的比。可以把这个数写成小数形式0.75，当写成小数时，它真正表示了分数，在此为 $\frac{75}{100}$ 。

回忆一下第7章里的小数系统，在小数点左边的数字与10的整数次幂相关联；同样，在小数点右边的数字与10的负整数次幂相关联。第7章用42 705.684作为例子，该数可以表示成与下面与之相等的形式：

$$4 \times 10\,000 +$$

$$2 \times 1000 +$$

$$7 \times 100 +$$

$$0 \times 10 +$$

$$5 \times 1 +$$

$$6 \div 10 +$$

$$8 \div 100 +$$

$$4 \div 1000$$

注意一下除号，可以把这个序列写成没有除号的形式：

$$4 \times 10\,000 +$$

$$2 \times 1000 +$$

$$7 \times 100 +$$

$$0 \times 10^+$$

$$5 \times 1 +$$

$$6 \times 0.1 +$$

$$8 \times 0.01 +$$

$$4 \times 0.001$$

最后，可以用 10 的幂的形式表示如下：

$$4 \times 10_4 +$$

$$2 \times 10_3 +$$

$$7 \times 10_2 +$$

$$0 \times 10_1 +$$

$$5 \times 10_0 +$$

$$6 \times 10_{-1} +$$

$$8 \times 10_{-2} +$$

$$4 \times 10_{-3}$$

有些分数并不容易用小数表示，常见的如 $1/3$ 。如果用 3 去除 1，可以得到：

$$0.33333333333333333333.....$$

而永无止境。我们通常写成简洁形式，在 3 上面加一道横线来表示无限循环：

0. —

3

即使这样，把 $1/3$ 写成小数也是有些笨拙的。它还是一个分数，因为它是两个整数的比。同样， $1/7$ 是：

0.142857142857142857..... 或 0.142857

无理数则更不同，如 2 的平方根。无理数不能表示成两个整数的比，也就是说，小数部分 是无穷的，没有重复规律或固定模式：

$\sqrt{2}=1.414213\ 56237309504880168872420969807856967187537695...$

2的平方根是下面这个代数方程的根：

$$x_2 - 2 = 0$$

如果一个数不是以整数为系数的代数方程的根，则称为超越数（所有的超越数为无理数，但并不是所有的无理数都是超越数）。超越数包括 π ，它是圆的周长与直径的比，近似值为：

3.1415926535897932846264338327950288419716939937511.....

另一个超越数是 e ，它是下面表达式：

当 n 趋近于无穷大时的近似值:

□

□ 1+

□

$$1 \leq n$$

$$1$$

$$n$$

2.71828182845904523536028747135266249775724709369
996...

到现在为止，谈到的所有数——有理数和无理数——统称为实数。这种定义用来与虚数相区分。虚数是负数的平方根，复数是由虚数和实数组成的。不管名称如何，虚数揭示了现实世界的奥秘，可以用来（例如）解决电子学的一些高级问题。

习惯上，我们把数看成是连续的。如果给出两个有理数，则可以找出一个数在这两个数中间。实际上，只需取平均值即可。但是，数字计算机不能处理连续事件。位不是 0 就是 1，

没有中间值。由于这一特性，数字计算机必须处理 离散值 。可以表示的离散值的个数直接与 可达到的二进制位数相关。例如：如果用 32位来存放正整数，则可以存放 0~4 294 967 295个 整数。如果需要存放 4.5这个数，则必须重新考虑一种方法并做一些改动。

小数可以表示成二进制吗？是的，可以。最容易的方法可能是二进制编码的十进制

（BCD）。前面第 19章讲到 BCD是十进制数的二进制编码，每一个十进制数字（ 0、1、2、3、 4、5、6、7、8和9）需要4位，如下表所示：

十进制数字	二进制数字
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

BCD码特别适用于用美元和美分表示的与钱数有关的计算机程序。银行和保险公司是两个典型的多与钱打交道的行业，对这类公司的计算机程序来说，许多分数只需要两个十进制数位。

通常1个字节存储两个 **BCD**数字，有时将这称为压缩**BCD**码。2的补码不与 **BCD**一起使用，因此，压缩 **BCD**码通常要有额外的一位 (称作符号位)来标明是正数还是负数。一个 **BCD**数存入整个字节比较方便，所以，小小的符号位通常需要牺牲 4位或8位的存储空间。

来看一个例子。假定计算机要处理的钱数不会超过正、负 1000 万，换句话说，需要表示

的钱数的范围从－ 9 999 999.99～9 999 999.99，则存储在内存的每一笔钱数需要用 5个字节来

表示。例如，－ 4 325 120.35用5个字节表示为：

00010100 00110010 01010001 00100000 00100101

或用十六进制表示为:

14h 32h 51h 20h 25h

注意最左边的 1 用来表示负数，即符号位。如果是正数，则该位为 0。每一个数字需要 4 位，从十六进制值中可以直接看到。

如果需要表示的数的范围从 $-99\,999\,999.99 \sim 99\,999\,999.99$ ，则需要 6 个字节——10 个数字占 5 个字节，另一个字节仅用来表示符号位。

这类存储和标记方法也称作定点格式，因为小数点通常固定在特定的位置——本例中，小数点在两个小数位之前。注意，实际上并没有什么东西与数一起存放用来标明小数点的位置。处理定点格式数的程序应该知道小数点在哪里。定点数可以有任意个小数位，在同一计算机程序里可以混用这些数字，但是对这些数进行算术运算的那部分程序必须知道小数点的位置。

定点格式只在知道这些数不会超过预先确定的内存单元，且没有太多小数位的场合比较适

用。在数可能很大或可能很小的场合定点格式完全不适用。假设保留一个内存区域用来存储以英尺为单位的距离，则存在的问题是距离可能超出范围。从地球到太阳的距离是 490 000 000 000

英尺，氢原子的半径为 0.00000000026英尺，则你需要 12字节的定点存储空间来容纳这些可能 很大也可能很小的数值。

如果你还记得科学家和工程师们喜欢用称为“科学记数法”的系统来表示数的话，你也

许已找到更好的存储此类数的方法。科学记数法特别适用于表示很大和很小的数，因为它采用10的幂方法从而不用写很长的一串0。采用科学记数法后，数字

490 000 000 000 写成 $4.9 \times 10_{11}$

数字

0.00000000026 写成 $2.6 \times 10_{-10}$

在这两个例子里，数字 4.9和2.6称作小数部分或首数，有时也称作有效数（尽管这个词更 适用于对数运算）。为了与计算机术语相协调，在这儿把科学记数法的这一部分称作有效数。指数部分是 10的幂。在第一个例子中，指数是 11；在第二个例子中，指数是 - 10。指数

用来指明有效数的小数点要移动的位数。

为方便起见，有效数通常大于或等于 1而小于10。尽管下面的数字是相等的：

$$4.9 \times 10_{11} = 49 \times 10_{10} = 490 \times 10_9 = 0.49 \times 10_{12} = 0.049 \times 10_{13}$$

但我们选用第一种格式。这种格式也称作科学记数法的规格化格式。注意，指数符号只是标明数的大小而并不表示数本身是正的还是负的。下面是用科学记

数法表示的两个负数的例子：

$-5.8125 \times 10_7$ 等于 $-58\,125\,000$

和

$-5.8125 \times 10_{-7}$ 等于 -0.00000058125

在计算机中，对应于定点表示法的是浮点表示法。浮点格式用来存储较小或较大的数比较理想，因为它是以科学记数法为基础的。但是，计算机中采用的浮点格式是用科学记数法表示的二进制数。这里首先要提到的是如何用二进制表示小数数字。

实际上，这比设想的要容易，在十进制表示中，小数点右边的数字具有10的负整数次幂；在二进制表示中，二进制小数点（也仅是一个点，看起来与十进制小数点一样）右边的数具有2的负整数次幂。例如，一个二进制数：

101.1101

可以用以下表达式转换成十进制:

除号可以用 2 的负整数次幂替换:

$$1 \times 4 +$$

$$0 \times 2 +$$

$$1 \times 1 +$$

$$1 \div 2 +$$

$$1 \div 4 +$$

$$0 \div 8 +$$

$$1 \div 16$$

$$1 \times 2_2 +$$

$$0 \times 2_1 +$$

$$1 \times 2_0 +$$

$$1 \times 2_{-1} +$$

$$1 \times 2_{-2} +$$

$$0 \times 2_{-3} +$$

$$1 \times 2_{-4}$$

或者，2的负整数次幂可以从 1 开始重复除以 2 来计算：

$$1 \times 4 +$$

$$0 \times 2 +$$

$$1 \times 1 +$$

$$1 \times 0.5 +$$

$$1 \times 0.25 +$$

$$0 \times 0.125 +$$

$$1 \times 0.0625$$

通过这些计算得到 101.1101 等效的十进制数 5.8125。在十进制科学记数法中，规格化有效数通常大于或等于 1 而小于 10。同样，二进制科学记

数法的规格化有效数也通常大于或等于 1 而小于 10（即十进制中的 2）。所以，按二进制科学记数法，数

101.1101 表示成 $1.011101 \times 2_2$

这个规则隐含了一件有趣的事实：通常二进制浮点数在二进制小数点的左边除了 1 以外再没有别的了。

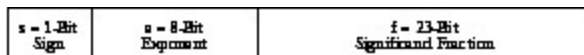
现代计算机和计算机程序按照 IEEE 在 1985 年制定的标准来处理浮点数，这个标准也为

ANSI(the American national standards institute，美国国家标准局)所认可。ANSI/IEEE Std 754-1985 称作 IEEE 二进制浮点数算术运算标准。它并不像一般标准那样长，只有 18 页，但却奠定了以方便的方式编码二进制浮点数的基础。

IEEE 浮点数标准定义了两个基本格式：单精度格式，需要 4 个字节；双精度格式，需要 8

个字节。

首先看一下单精度格式，它有三部分：1 位符号位（0 表示正，1 表示负）、8 位的指数位和 23 位的有效数位。如下所示，最低有效数在最右边：



s = 1 位符号 e = 8 位指数 f = 23 位有效数

总共有 32 位，4 个字节。因为规格化二进制浮点数的有效数通常在二进制小数点左边为 1，所以在 IEEE 格式中这一位不包含在浮点数的存储空间中。有效数的 23 位小数部分是反被存储的部分，所

以，即使只有 23 位用来存储有效数，精度仍然认为是 24 位的。过一会儿将要看到 24 位精度的意义。

8 位指数范围从 0~255，称为 移码指数，意思是必须从指数中减去一个数（称为 偏移量）

才能确定有符号指数的实际值。对单精度浮点数，偏移量为 127。

指数0和255用于特殊用途，在此简单描述一下。如果指数从 1 变化到254，则由s（符号位）、e（指数）和f（有效数）来表示的数为：

$$(-1)_s \times 1.f \times 2^{e-127}$$

-1 的 s 次幂是数学上的一种方法，意思是“如果 s 为0，则数是正的（因为任何数的 0 次幂 等于1）；如果 s 为1，则数是负的（因为 -1 的1次幂为 -1 ）”。

表达式的另一部分是 $1.f$ ，意思是 1 后面为二进制小数点，再后面为 23 位的有效小数部分。它乘以2的幂，其中指数为内存中的 8 位移码指数减去 127。

注意，到现在还没有提到如何表示一个很常见的数字，那就是 0。这是一种特殊情况，即：

- 如果 e 等于0，且 f 等于0，则数为0。通常，所有32位均为0则表示0。但是符号位可以是 1，在这种情况下，数被解释为 -0 。 -0 可以表示一个很小的数，小到在单精度格式中不能用数字和指数来表示。尽管如此，它们然小于 0。

- 如果 e 等于0，且 f 不等于0，则数是有效的。但是，它不是规格化的数，它等于

$$(-1)_s \times 0.f \times 2^{-127}$$

注意，二进制小数点左边的有效数为 0。

- 如果 e 等于255，且 f 等于0，则数为正或负无穷大，这取决于符号 s 。

- 如果 e 等于255，且 f 不等于0，该值被认为“不是一个数”，简写为 NaN。NaN 可以表示一个不知道的数或者一个无效操作的结果。

通常，单精度浮点格式中可以表示的最小规格化的正或负二进制数为：

1.000000000000000000000000

TWO

×2₋₁₂₆

在二进制小数点之后有 23个0。在单精度浮点格式中可以表示的最大规格化的正或负二进制数为：

1.11111111111111111111

TWO

$\times 2_{127}$

换算或十进制，这两个数近似为 $1.175494351 \times 10^{-38}$ 和 $3.402823466 \times 10^{38}$ 。这就是单精度浮点数表示法的有效范围。

前面讲过，10位二进制数近似等于3位十进制数。也就是说，若10位都置1(即十六进制为

3FFh，十进制为1023)，则它近似等于3位十进制都设置为9，即999。或者

$$2_{10} \approx 10_3$$

这种关系表明按单精度浮点格式存放的24位二进制数大约与7位十进制数等效。因此，也可以说单精度浮点格式提供24位二进制精度，或大约7位十进制精度。它的含义是什么呢？

当观察定点数的时候，数的精度是很显然的。例如，对于钱数，用两位十进制小数的定点数就可精确到分。但是，对浮点数来说，就不能这么肯定了。根据指数值的不同，有时浮点数可以精确到比分还小的单位，有时甚至不能精确到元。

粗略地讲，单精度浮点数可精确到 $1/2^{24}$ ，或 $1/16777216$ ，或约百万分之六。这到底是什么意思呢？

从某种意义上讲，它意味着如果想用单精度浮点数来表示16 777 216和16 777 217，其结

果是一样的。而且，在这两个数之间的任何数（如16 777 216.5）也认为是与它们一样的。所

有这3个十进制数都按 32位单精度浮点数

4B800000h

来存放。当把此数分成符号位，指数和有效数位时，如下所示：

0 10010111 000000000000000000000000

也即为

1.000000000000000000000000

TWO

$$\times 2_{24}$$

下一个表示的最大有效数是 16 777 218，它的二进制浮点表示为：

1.000000000000000000000001

TWO

$$\times 2_{24}$$

两个不同的十进制数却以相同的浮点数存放可能是也可能不是一个问题。如果是为银行编写程序且用单精度浮点数来存储元、分等，则你可能会很苦恼地发现

\$262 144.00与\$262 144.01是一样的。两个数字都是：

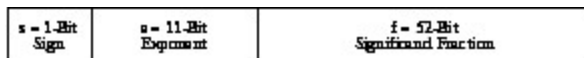
1.000000000000000000000000

TWO

$\times 2_{18}$

这就是当处理元、分的时候，为什么要用定点数的原因。在处理浮点数的时候，可能还会发现其他足以使人发疯的小毛病。程序原本计算的结果是 3.50却成了3.499999999999。浮点计算中这种事情经常发生，但也没有别的更好的处理方法。

如果想用浮点表示法，又不想出现单精度那样的问题，可以用双精度浮点格式。这样的数需要8个字节来存放，格式如下：



s = 1 位符号 e = 11 位指数 f = 52 位有效数

指数偏移量为 1023，即3FFh，所以，以这种格式存放的数为

$(-1)_s \times 1.f \times 2^{e-1023}$ 它具有与单精度格式中所提到适用于 0、无穷大和 NaN等情形相同的规则。

最小的双精度浮点格式的正数或负数为

最大的数为

(1.0.....0)

52↑0

TWO

$$\times 2_{-1022}$$

$\underbrace{\hspace{1.5cm}}$

$(1.1.....1)$

$52\uparrow 1$

TWO

$\times 2_{1023}$

用十进制表示，它的范围近似为 $2.2250738585072014 \times 10^{-308} \sim 1.7976931348623158 \times$

10_{308} 。10的308次幂是一个非常大的数，在1后面有308个十进制零。

53位有效数（包括没有包含在内的那1位）的精度与16个十进制位表示的精度十分接近。相对于单精度浮点数来说这种表示要好多了，但它仍然意味着最终还是有一些数与另一些数是相等的。例如，140 737 488 355 328.00与140 737 488 355 328.01是相同的，这两个数按照64位双精度浮点格式存储，结果都是：

可把它转换为：

42E0000000000000h



(1.0.....0)

52个0

47

×2

TWO

当然，开发一种格式用来在存储器中存储浮点数只是在汇编语言程序中实际使用这些数的工作中的一小部分。如果真的要研制与世隔绝的计算机，则你需要面对编写浮点数的加、减、乘、除的函数集的工作。幸运的是，这些工作可以被分解成许多小的只涉及到整数的加、减、乘、除的工作，而整数的四则运算我们已经知道如何实现了。

例如，浮点加法的关键是有效数相加，因而用的技巧是用两个数的指数部分确定有效数如何移位。假设要做以下加法：

$$(1.1101 \times 2_5) + (1.0010 \times 2_2)$$

需要把 11101 与 10010 相加，但不是就这样相加。指数部分的不同表明第二个数必须进行移位。实际上，需要进行 11101000 和 10010 的整数加法。最后的和是：

$$1.1111010 \times 2_5$$

有时两个数的指数部分差距很大，其中一个数甚至对和没有影响。就像这种情况：把地球到太阳的距离与氢原子的半径相加。

两个浮点数的相乘是把有效数部分像整型数那样相乘并把两个整型指数相加。通常，规格化有效数部分可能会引起对新的指数调整一、二次。

浮点算术运算中另一个复杂问题牵涉到较麻烦的计算，如方根、幂、对数和三角函数。

但是，所有这些工作都可以用四个基本的浮点操作：加、减、乘、除来完成。例如，三角函数 Sin 可以通过下列展开式来计算，如下：

参数 x 必须是弧度，360度的弧度为 2π 。感叹号是阶乘符号，其含义是把1到该数之间的所有整数相乘，如： $5! = 1 \times 2 \times 3 \times 4 \times 5$ 。这只是进行乘法运算，其中每一项的指数部分也是乘法。其余的是一些除法、加法和减法。唯一真正麻烦的部分是在最后的省略，它意味着要永远地计算下去。然而，实际上，如果局限在 $0 \sim \pi/2$ 的范围（从这里可以推导出所有其他的正弦函数值），并不需要进行多少展开运算。在展开大约12项以后，已经精确到了双精度数的53位。

当然，使用计算机是为了使人们更容易完成某些工作，所以，编写浮点运算程序这样的工作似乎离使用计算机的目的相差甚远。然而，这正是软件的可爱之处：一旦某人为某台机器编写了浮点运算程序，其他人都可以使用。对科学和工程应用程序来说，浮点运算非常重要，所以通常有很高的优先权。在计算机出现的早期，一旦新的类型的计算机出来，编写浮点运算程序通常是第1项软件工作。

事实上，甚至可以设计计算机机器码指令直接进行浮点运算！显然，说起来容易做起来难，但这也说明了浮点运算的重要性。如果可以用硬件来实现浮点运算——与16位微处理器的乘法和除法指令一样——则计算机中所有浮点运算工作将会完成得更快。

最早把浮点运算硬件作为选件的商用计算机是1954年的IBM 704，704把所有的数按36位来存储。对浮点数，分成27位的有效数、8位指数和1个符号位。浮点运算硬件可做加法、减法、乘法和除法，其他浮点运算功能必须用软件来实现。

桌面机的浮点运算硬件出现在 1980 年，当时 Intel 发布了 8087 数字数据协处理器芯片，一种集成电路芯片，今天通常称为数学协处理器或浮点运算单元（floating-point unit，FPU）。8087 之所以称为协处理器是因为它不能自己单独使用，它只能与 8086 或 8088 一起使用，8086 和 8088 是 Intel 的第一个 16 位微处理器。

8087 有 40 个引脚，使用许多与 8086 和 8088 相同的信号。微处理器和数学协处理器通过这些信号连接起来。当 CPU 收到一个特殊指令——称为 ESC，代表 **Escape**——则协处理器接管系统控制权并执行下一条机器代码，即包括三角运算、指数、对数运算的 68 条指令中的一条。数据类型以 IEEE 标准为基础。那时，8087 被认为是所生产的最高级的集成电路。

可以认为协处理器是一个小的自包含的计算机。在响应某个浮点运算机器码指令时（例如，计算平方根的 FSQRT 指令），协处理器内部执行存放在 ROM 中的自己的指令序列，这些内部指令称为微代码。这些指令通常是循环的，所以计算结果并不是马上可用。尽管如此，一般来说，数学协处理器至少比用软件来实现的同样例程要快 10 倍。

初始的 IBM PC 主板在 8088 芯片的右边有一个 40 管脚的插槽供 8087 用。遗憾的是，这个插槽是空的，需要加速浮点运算的用户必须单独购买 8087 并自己把它安装上。即使在安装了数学协处理器后，并不是所有的应用程序都可以运行得更快，一些应用程序——如，文字处理程序——几乎不需要浮点运算。其他如电子报表程序则要用到很多浮点计算。这些程序能够运行得更快，但并不是所有程序都是如此。

可以看到，程序员必须用协处理器机器码指令来编写特定的代码供协处理器执行。因为数学协处理器不是硬件的标准部分，因而许多程序员怕麻烦不愿意做。但是，他们还是不得不编写自己的浮点运算子程序（因为许多人并没有安装数学协处理器），所以支持 8087 芯片就成为一个额外的负担——一个不小的负担。最

终，如果他们程序运行的机器上有数学协处理器，程序员要学会编写利用数学协处理器的应用程序；如果没有，则要编写浮点运算仿真程序。

经过几年后，Intel还发布了用于286芯片的287数学协处理器，用于386的387数学协处理器。但对于1989年发布的Intel 486DX，FPU已经做在了CPU里面，而不再是作为一个选件！遗憾的是，1991年Intel发布了一种低价格的486SX，它没有把FPU做在CPU里面，而是提供了487SX数学协处理器作为一个选件。1993年发布的Pentium芯片却再一次使做在CPU内部的FPU成为标准，也许以后永远会这样。Motorola在它的68040微处理器里集成了FPU，该微处理器于1990年发布。以前，Motorola销售68881和68882数学协处理器用来支持早先68000家族的微处理器。PowerPC芯片也把浮点运算硬件集成在内部。

尽管浮点运算硬件对专门从事汇编语言程序设计的程序员来说是一个很好的礼物，但是，与20世纪50年代早期开始的其他一些工作相比这只是微不足道的进步。我们的下一个主题是：计算机语言。

第 24 章 高级语言和低级语言

用机器码编程就像用牙签吃东西，刺的块很小且做起来很费力，吃一顿饭要花很长时间。同样，每个机器码字节只是完成可以想像得到的最小且最简单的计算工作——从内存装入一个数至处理器，与把它另一个数相加，再把结果存回到内存——所以，很难想像机器码如何完成一整项的工作。

至此，我们至少已从第 22 章开始处的原始模型阶段有了一些进步，从前我们一直用控制

面板上的开关输入二进制数据到内存。在第 22 章里，介绍了如何写简单的程序用键盘输入并在视频显示器上检查机器码的十六进制字节码。这当然不错，但还不是改进的终点。

正如我们所知道的，机器码字节与某些短的助记符相关联，如 **MOV**、**ADD**、**CALL** 和 **HLT**，因此，可以用一些模糊类似的英语来引用机器码。这些助记符通常与操作数写在一起，进一步表明机器码指令的功能。例如：8080 机器码字节 46h 使得微处理器把寄存器对 **HL** 中的 16 位数寻址的内存单元所存放的内容传送到寄存器 **B**。这可以很简明地写成：

```
MOV B, [HL]
```

当然，用汇编语言编写程序比用机器码要容易得多，但微处理器并不能理解汇编语言。我们已经讲过如何在纸上编写汇编程序，只有当想在微处理器上运行汇编语言程序的时候，才会手工汇编程序，意思是把汇编语言语句转换成机器码字节并输入到内存。

如果能让计算机来做这项转换工作就更好了。如果你正在 8080 计算机上运行 CP/M 操作系统，则你已经具有了所需要的工具。下面介绍它是如何工作的。

首先，建立一个文本文件，包含有你用汇编语言编写的程序。可以用 CP/M 程序 ED.COM 来完成这项工作。该程序是一个文本编辑器，可用来创建、修改文本文件。现假设创建的文本文件名称为 PROGRAM1.ASM。ASM 文件类型表明该文件是汇编语言程序文件，该文件看起来就像下面这样：

```
ORG 0100h LXI DE, Text MVI C,9

CALL 5 RET

Text: DB 'Hello!$'

END
```

文件中有几个语句以前没有见过。第一个是 ORG (origin) 语句，该语句不对应于任何一条 8080 指令，它表示下一条语句的地址从地址 0100h 处开始。前面讲过，这就是 CP/M 程序装入到内存的地址。

下一个语句是 LXI (load extended immediate) 指令，它装入一个 16 位数到寄存器对 DE。本例中，16 位数由标号 Text 给出。标号在程序的下部，DB (Data Byte) 语句之前。DB 也是以前未见到的，DB 语句后面可以是几个逗号隔开的字节或（就像本例中）在单引号里的一些字符。

MVI (**move immediate**) 语句把数 9 送到寄存器 **C**。**CALL 5** 语句进行 **CP/M** 功能调用。功能 9 的意思是：显示一个字符串，起始地址由寄存器对 **DE** 给出，遇到美元符号结束。（注意，文本以美元符号作为字符串的结束是很奇怪的，但 **CP/M** 就采用这种方法。）最后的 **RET** 语句用来结束程序并把控制权返还给 **CP/M**（这实际上是结束 **CP/M** 程序的几种方法之一）。**END** 语句表示汇编语言文件结束。

既然已经有了 7 行文本的文本文件，下一步是汇编该文件，即把它转换成机器码。以前是用手工来完成，自从运行 **CP/M** 后，可以用包含在 **CP/M** 中的名为 **ASM.COM** 的程序来完成。这个程序是 **CP/M** 汇编程序，在 **CP/M** 命令行运行 **ASM.COM**，方法为：

```
ASM      PROGRAM1.ASM
```

ASM 程序汇编 **PROGRAM1.ASM** 文件并创建新的文件，名为 **PROGRAM1.COM**，它含有与编写的汇编语言程序相对应的机器码（实际上，在这个过程中还有另外一步，但在这里并不重要）。现在，在 **CP/M** 命令行就可以运行 **PROGRAM1.COM**，结果显示字符“**Hello!**”，然后结束。

PROGRAM1.COM 文件包含有下面 16 个字节：

```
11 09 01 0E 09 CD 05 00 C9 48 65 6C 6C      6F 21 24
```

前面 3 个字节是 **LXI** 指令，紧接着 2 个字节是 **MVI** 指令，再后面 3 个字节是 **CALL** 指令，然后是

RET 指令，最后 7 个字节是“**Hello**”、感叹号和美元符号的 **ASCII** 码。像 **ASM.COM** 这样的汇编程序所做的工作是：读入一个汇编语言程序（常称作源代码文件），

产生一个包含有机码的文件——可执行文件。从大的方面来看，汇编程序是相当简单的程序，因为在汇编语言助记符与机器码之

间存在一一对应的关系。汇编程序把每一个文本行分成助记符和参数，然后把这些单词和字符与一张表相对照，该表中存有所有可能的助记符和参数。通过这种对照就可以找到每个语句所对应的机器码指令。

注意汇编程序是如何得出 **LXI**指令必须把寄存器对 **DE**设置为地址 **0109h**的。如果 **LXI**指令本身在 **0100h**处（**CP/M**把程序装入内存运行时的地址），则**0109h**是就**Text**字符串的开始地址。通常，使用汇编程序的程序员并不需要关心程序各部分的地址。

当然，第一个编写汇编程序的人必须手工汇编程序。在同一台计算机上编写新的（或改进）汇编程序的人可以用汇编语言编程然后用最初的汇编程序来汇编。一旦新的汇编程序经过了汇编，它也可用来汇编自身。

每当一个新的微处理器诞生，就需要新的汇编程序。新的汇编程序可以在已有的计算机上编写，利用原有的汇编程序来汇编。这种汇编称之为交叉汇编，即用在计算机 **A**上的汇编程序来生成在计算机 **B**上运行的代码。

尽管汇编程序消除了汇编语言编程缺少创造性这一问题（手工汇编部分），但汇编语言还存在两个主要问题，第一个（也许你已经猜测到了）是汇编语言程序冗长、乏味。因为你是 在微处理器芯片级编程，所以必须要考虑每一个细节。

第二个问题是汇编语言不可移植。如果为 **Intel 8080** 编写汇编语言程序，则不适用于在 **Motorola**的**6800**上运行，必须用 **6800**的汇编语言重新编程。也许，这不像编写最初的汇编语言程序那么困难，因为已经解决了主要的组织和算法问题，但是，仍然有许多工作要做。

上一章解释了现代微处理器芯片如何集成机器码指令来进行浮点运算，这当然已经很方便了，但还不是十分令人满意。一种选择是彻底放弃与处理器相关的实现每个基本算术操作

的机器码，取而代之的是用代数符号来表示许多数学运算。以下是一个例子：

$$A \times \sin (2 \times \pi + B) / C$$

这里A、B和C是数字， $\pi = 3.14159$ 。既然如此，何乐而不为呢？如果这样的一条语句是在一个文本文件里，则可以编写汇编

语言程序读取文本文件并把代数表达式转换成机器代码。如果只计算一次这样的代数表达式，则可以手算或用计算器计算。如果需要不同的A、

B、C值来计算表达式的值，则可能需要考虑如何用计算机来计算。正因为如此，代数表达式不可能单独出现，应该考虑到表达式的上下文，用不同的值代入计算。

现在已开始创建所谓的高级程序设计语言。汇编语言称作低级语言，因为它与计算机硬件密切相关。尽管“高级”用来描述除汇编语言以外的任何程序设计语言，但这些语言中，一些语言还比另一些语言更要高级一些。如果你是一家公司的总裁，且坐在计算机前输入“计算全年的收益和支出，做出年度报表，打印两千份给所有的股东”，那么你确实正在用非常高级的语言工作。在现实生活中，程序设计语言并没有达到这样理想的境界。

人类语言是千百年来复杂的影响、随机变化和不断适应的结果，即使像世界语这样的人工语言也来源于现实语言。然而，高级计算机语言是审慎而周密的概念语言。发明程序设计语言面临的挑战是如何使语言具有吸引力，因为语言定义了人们如何向计算机发送指令。从20世纪50年代开始到1993年，估计已发明和实现了1000多种高级语言。

当然，这还并不足以简单地定义高级语言（它牵涉到语言所采用的语法），还必须有编译程序用来将高级语言语句转换成机器

码。像汇编程序一样，编译程序需要一个字符接一个字符地读取源代码文件，并分解成短语、符号和数字，但编译程序比汇编程序更复杂。从某种意义上讲，汇编程序较简单，因为在汇编语言语句与机器码之间有一一对应的关系。编译程序通常要把一条高级语言语句转换成许多机器码指令。编译程序不容易编写，许多书中描述了它们的设计与构造，所以本书不作介绍了。

高级语言有优点也有缺点。最主要的优点是高级语言比汇编语言容易学且容易编写。用高级语言编写的程序清晰、简明。高级语言通常是可移植的——也就是说，它不像汇编语言那样依赖于特定的处理器。所以，程序设计员不需要知道程序将要运行其上的机器的内部结构。当然，如果需要通过程序在不止一种处理器上运行，则需要相应的编译程序生成针对这些处理器的机器码。可执行文件仍然只适用于某一个处理器。

另一方面，差不多都是如此，一个好的汇编语言程序设计员可以写出比编译程序所能产生的更优化的代码。也就是说，用高级语言编写的程序所产生的可执行文件比用汇编语言编写功能相同的程序所产生的可执行文件要大，且执行速度较慢。（最近几年，随着微处理器的日趋复杂以及编译程序在优化代码方面的日趋成熟，这种差别已变得不很明显。）

还有，尽管高级语言使得处理器更容易使用，但并没有使它的功能更强大。而使用汇编语言可以最大限度地利用处理器的能力。因为高级语言需要转换成机器码，所以高级语言只会降低处理器的能力。如果一个高级语言是真正可移植的，则它不能使用某种处理器的独有特点。

例如，许多处理器都有移位指令。前面讲过，这些指令把累加器中的位向左或向右移动。但是，几乎没有高级语言有这样的操作。如果程序中要用到移位，则需要用乘2或除2操作来

处理（其实，现在许多编译程序都用处理器的移位指令来实现乘或除以2的幂）。许多高级语

言同样也不包括按位的逻辑运算。早先的家用计算机中，许多应用程序是用汇编语言编写的，而现在除非有特殊需要，汇

编语言已经很少用到。由于已在处理器中添加了一些硬件来实现流水线技术——同时有多个指令码累进执行——汇编语言则变得更难以掌握。与此同时，编译程序却逐步走向成熟。现

代计算机的大容量存储能力也在这种趋势——程序设计员不再满足于编制在小的内存和磁盘

上运行的代码——中也扮演着重要角色。尽管许多早期计算机的设计者都试图用代数符号来阐明他们的观点，但通常认为第一个

真正成功的编译程序是由 Grace Murray Hopper（1906—1992）于 1952 年在雷明顿为 UNIVAC

而设计的 A-0。当 Hopper 博士 1944 年为 Howard Aiken 工作时，就已开始了计算机的早期研究工作。在她 80 多岁时，仍然活跃在计算机界，为 DEC 公司作一些公关工作。

今天仍然在使用的最古老的高级语言（尽管这些年中得到了广泛的修改）是 FORTRAN。许多计算机语言的名字都是大写字母，因为它们是由许多单词的首字母构成的。FORTRAN 是由 FORMula 前 3 个字母和 TRANslation 的前 4 个字母混合而成，是在 20 世纪 50 年代中期由 IBM 公司为 704 系列计算机开发的。多年来，FORTRAN 一直被选作为科学和工程的计算语言，它

广泛支持浮点运算甚至支持复数运算。

所有计算机语言都有它们的支持者和批评者，并且人们可能只热衷于他们所喜爱的语言。尽量站在中立的立场上，我选择一种语言作为原型来解释那些差不多再没有人用的程序设计概念。这种语言的名字是 ALGOL（即 ALGOrithmic Language）。ALGOL 也

可用来探索高级程序设计语言的本质，因为从某种意义上来说它正如一粒种子，成为过去 40 年来许多流行的通用语言的直接祖先。甚至在今天，人们也用到“类 ALGOL”的程序设计语言。

它的第一个版本是 ALGOL58，由一个国际委员会在 1957~1958 年设计而的。两年后，即 1960 年进行了修改，修订版命名为 ALGOL 60。最后的版本是 ALGOL 68。本章用到的 ALGOL 版本在文档“Revised Report on the Algorithmic Language ALGOL 60”中有描述，该文档在 1962 年完成，1963 年第 1 次印刷。

让我们来编写一些 ALGOL 代码。假设一个名为 ALGOL.COM 的编译程序运行在 CP/M 或

MS-DOS 下。第一个 ALGOL 程序是一个名为 FIRST.ALG 的文本文件，注意它的文件类型。一个 ALGOL 程序必须由 begin 和 end 作为开始和结束，以下为显示一行文本的程序：

```
begin  
  
print('This is my fist ALGOL program!'); ende
```

可以用 ALGOL 编译程序来编译 FIRST.ALG 程序，操作如下：

```
ALGOL FIRST.ALG
```

ALGOL 编译程序的响应可能是显示类似于下面的内容：

```
Line 3: Unrecognized keyword 'ende'.
```

ALGOL 对拼写的挑剔不亚于传统的英语教师。在输入程序时若拼错了单词 end，编译程序则会告知程序有一个语法错误。当它碰到 ende 时，它希望那是它可以识别的关键字。

修改了错误以后，可以再运行 ALGOL 编译程序。有时，编译程序会直接生成一个可执行文件（名为 FIRST.COM，或者是 MS-DOS 下的 FIRST.EXE）；有时，还需要进行另一个步骤。无论怎样，你都可以从命令行运行 FIRST 程序：

FIRST

FIRST程序的响应是显示:

```
This is my fist ALGOL program!
```

糟糕! 还有一个拼写错误。这是一个编译程序不能发现的错误, 因此, 称为运行时错误

(run-time error) —即只在运行程序时才出现的错误。

可以看出, 在该 ALGOL 程序中, `print` 语句在屏幕上显示一些内容, 本例是一行文本 (因此, 这个 ALGOL 程序等效于本章前面 CP/M 下的汇编程序)。`print` 语句实际上并不是 ALGOL 语言正式定义的一部分, 这里只假设正在用的这个 ALGOL 编译程序包含有这样一个实用工具, 有时称作内部函数。`print` 语句——就像许多 ALGOL 语句 (除 `begin` 和 `end` 外) 一样——后面必须跟引号。`print` 语句向里缩进不是必须的, 只不过使得程序结构更清晰。

假设要编写一个程序计算两个数的乘法。每一个程序设计语言都有变量这个概念。在程序中, 变量名可以为一个字母、一个短的字母序列, 甚至为一个短词。实际上, 变量对应于一个内存单元, 但在程序中是通过名字来引用的, 并不是通过内存地址。下面这个程序有 3 个变量, 名为 `a`、`b` 和 `c`:

```
begin

real a,b,c; a:=535.43; b:=289.771;

c:=a × b;

print ('The product of ', a, ' and ', b, ' is ', c);

end
```

`real` 语句是说明语句, 用来表明程序中要说明的变量。本例中, 变量 `a`、`b`、`c` 是实数或浮点数 (ALGOL 也支持关键字 `integer`,

用来说明整型变量)。通常, 程序设计语言要求变量名以字母开头。只要第一个字符是字母, 变量名可以包含数字, 但不能包含空格及许多其他字符。通常编译程序要限制变量名的长度。本章的例子都采用一个字母作为变量名。

如果使用的 **ALGOL** 编译程序支持 **IEEE** 浮点数标准, 则程序中的 3 个变量都需要 4 个字节的存储空间 (对单精度数) 或 8 个字节的存储空间 (对双精度数)。

接下来的三个语句是赋值语句。在 **ALGOL** 中, 赋值语句定义为冒号后紧跟等号。(在许多计算机语言中, 赋值语句只需用等号。) 赋值语句的左边是变量, 右边是表达式。前两个赋值语句是给 **a** 和 **b** 赋给一个值, 第三个赋值语句中变量 **c** 的值由变量 **a** 和 **b** 产生。

今天, 在程序设计语言中, 大家熟悉的 \times (乘号) 通常不允许使用, 因为它不属于 **ASCII** 码和 **EBCDIC** 的字符集。许多程序设计语言用星号 ($*$) 表示乘法。虽然 **ALGOL** 用斜杠 ($/$) 表示除法, 但也包括一个除号 (\div) 表示整数除法, 即表明被除数中有多少倍的除数。**ALGOL** 中也定义了箭头 (\uparrow), 这是另一个非 **ASCII** 码字符, 用来表示乘方。

最后是用来显示的 **print** 语句。本例中即有文本又有变量, 它们用逗号隔开。显示 **ASCII** 字符可能并不是 **print** 语句的主要工作, 本例中, 它的功能还包括把浮点数转换成 **ASCII** 码:

```
The product of 535.43 and 289.711 is 155152.08653
```

接着程序终止, 返回到操作系统。如果想乘另外两个数, 则需要修改程序, 改变数, 重新编译, 再运行。可以利用一个名

为 **read** 的内置函数来避免这种频繁的重新编译工作:

```
begin
```

```
real a,b,c;
```

```
print ('Enter the first number: '); read (a);
```

```
print ('Enter the second number: '); read (b);
```

```
c:= a × b;
```

end

```
print ('The product of ', a, ' and ', b, ' is ', c);
```


`read`语句从键盘读入 `ASCII`码字符并转换成浮点数。高级语言中一个非常重要的结构是循环。循环使得同一段程序依据一个变量的多个不同

的值来运行。假设有一段程序用来计算 3、5、7和9的立方，就可以这样做：

```
begin
```

```
real a, b;
```

```
for a := 3, 5, 7, 9 do begin
```

```
    b := a × a × a;
```

```
    print (' The cube of ', a, ' is ', b);
```

end

end

for语句设置变量 **a**的初值为 3，然后执行 **do**关键字以后的语句。如果要执行的语句不止一条

（本例中正是如此），则这些语句必须包括在 **begin**和**end**之间，这两个关键字定义了一个语句块。**for**语句接着把变量 **a**设置成 5、7和9，并执行这些相同的语句。

下面是**for**语句的另一种形式，它计算 3~99间奇数的立方值：

```
begin
```

```
real a, b;
```

```
for a :=3 step 2 until 99 do begin
```

```
    b := a × a × a;
```

```
    print ('The cube of ', a, ' is ', b);
```

end

end

`for`语句设置变量 `a` 的初值为 3，然后执行 `for`语句后的语句块。然后 `a`以`step`关键字后面的值 2为 步长增加，得到新值 5，并用来执行代码块。变量 `a`不断加2，当它超过 99时，`for`循环结束。

程序设计语言通常都有非常严格的语法。例如，在 `ALGOL 60` 中，关键字 `for`后只能跟一种类型的东西，即变量名。而在英语里，单词 `for`后可以跟许多不同的单词，如“`for example`”。虽然编译程序不是容易编写的简单程序，但它显然要比解释人类语言的程序要简单得多了。

大多数程序设计语言的另一个重要特性是包含条件语句。条件语句只是在某个条件为真时才允许执行另一条语句。下面是使用 `ALGOL`内部函数 `sqrt`的一个例子，用来计算平方根。`sqrt`函数不能用来处理负数，所以程序中应避免出现这种情况：

```
begin
```

```
real a, b;
```

```
print ('Enter a number: '); read (a);
```

```
if a < 0 then
```

```
print ('Sorry, the number was negative.');
```

```
begin
```

```
    b = sqrt (a);
```

```
    print ('The square root of ', a, ' is ', b);
```


end

end

左尖括号 $<$ 是小于号。如果用户输入的一个数小于 0，则执行第一个 **print** 语句。否则，该数大于等于 0，则执行包含另一个 **print** 语句的语句块。

到目前为止，本程序中的每个变量只能存放一个值。用一个变量来存放多个值也是很方便的，这就是数组。ALGOL 程序中声明一个数组的方法如下所示：

```
real array a[1:100];
```

本例中，表明要用该变量来存储 100 个不同的浮点值，这些值称作数组元素。第一个为

a[1]，第二个为 **a[2]**，最后一个为 **a[100]**。方括号中的数字称作数组下标。下例程序计算从 1~100 的所有数的平方根，把结果存放在数组中并显示出来：

```
begin
```

```
real array a[1:100]; integer i;
```

```
for i :=1 step 1 until 100 do a[i] := sqrt(i);
```

```
for      i :=1 step 1 until 100 do
```

```
    print ('The square root of ', i, ' is ', a[i]);
```

```
end
```

程序中也声明了一个整型变量，名为 **i**（因为它是 **integer** 的第一个字母，所以经常用来作为整型变量名）。在第一个 **for** 循环

中，数组的每一个元素赋值为它的下标的平方根；第二个 **for** 循环中，输出这些值。

除了实型和整型外，变量还可以声明为布尔型（为了纪念第 10 章提到的乔治·布尔）。一个布尔变量只有两个可能的值，即 **true** 和 **false**。本章的最后一个程序里将用到布尔数组（和到目前为止学到的几乎所有特性）。该程序实现称为“**Eratosthenes**漏勺”的用来找到素数的著名算法。**Eratosthenes**（大约公元前 276-196 年）是亚历山大传说中的图书馆的管理员，他由于精确地计算出了地球的圆周长而名垂史册。

素数是指只能被 1 和它本身整除的自然数。第一个素数是 2（唯一的偶数素数），此外，素数还有 3、5、7、11、13、17 等等。

Eratosthenes 方法是从以 2 开始的正的自然数列表开始。因为 2 是素数，则要删除所有是 2

的倍数的数（即除 2 以外的所有偶数），这些数都不是素数。因为 3 是素数，则要删除所有是 3 的倍数的数。已经知道 4 不是素数，因为它已被删除了。下一个素数是 5，则要删除所有是 5 的倍数的数。依此类推，那些余下的数就是素数。

下面的 ALGOL 程序用来确定 2~10 000 的所有素数，通过声明一个布尔数组来标识从 2 ~

10000 的所有数来实现该算法：

```
begin

Boolean array a[2:10000]; integer      i, j

for i :=2 step 1 until 10000 do a[i] := true;


for i :=2 step 1 until 100 do if a[i] then

    for j := 2 step 1 until 10000 ÷ i do

        a [i × j] := false;
```

end

```
for i := 2 step 1 until 10000 do if a[i] then
```

```
  print (i);
```

第一个 `for` 循环把数组所有元素的布尔值设置为 `true`。这样，程序一开始假设所有的数都是素数。第二个 `for` 循环从 1~100（为 10 000 的平方根）。如果数是素数，意味着 `a[i]` 为真，则另一个 `for` 循环用来把该数的倍数设置为 `false`，这些数都不是素数。最后一个 `for` 循环输出所有的素数，即 `a[i]` 为真时对应的 `i` 值。

有时人们在争论程序设计到底是一门艺术还是一门科学。一方面需要在大学里学习有关计算机科学的课程，另一方面又要看著名的如 Donald Knuth 的《The Art of Computer Programmign》系列这样的书。物理学家 Richard Feynman 写道“更确切的说，计算机科学更像工程——都是用一些东西来实现另一些东西”。

如果让 100 个不同的人来编写输出素数的程序，将会得到 100 个不同的方法。即使这些程序员都有“Eratosthens 漏勺”这种思想，也不会正好以同样的方法实现。如果程序设计真的是一门科学，就不会有如此多的方法，而不正确的解决方法也是经常有的。偶尔程序设计问题会激起富有创造性和敏锐观察力的火花，而这也就是“艺术”的成分。但是，程序设计更多的是设计和组装的过程，就像在架设一座大桥。

早期的许多程序设计员都是科学家和工程师，他们利用 FORTRAN 和 ALGOL 所要求的数学算法来阐述自己的问题。然而，纵观程序设计语言的历史可以发现，人们希望有能被更大范围的人们所使用的语言。

第一个为商务系统设计的成功语言是 COBOL（common business oriented language），今天仍被广泛使用。由美国工业和国防部组成的委员会于 1959 年早期推出了 COBOL，它受到了 Grace Hopper 的早期编译程序的影响。从某种意义上说，COBOL 使得管理人员——可能并不具体设计编码——至少可以看懂程序代码并且能够检查代码是否按所预定的去工作（在现实生活中，这种情况很少发生）。

COBOL 广泛支持记录和生成报表。记录是按照一致方式组织的信息的集合体，例如：保

险公司可能要维持包含有它所卖的所有险种的一个大文件，每一险种为一单独记录，包括客 户姓名、出生日期和其他信息。早期的许多 COBOL 程序设计成能处理存储在 IBM 穿孔卡片上 的80列记录，为了尽可能少地占用卡片空间，日期中的年份通常用 2位编码而不是 4位，这导 致了随着 2000年的到来而普遍出现的“千年虫”问题。

20世纪60年代中期，伴随着 System/360项目的开发， IBM公司开发了名为 PL/I的程序设计 语言（ I是罗马数字 1， PL/I表示 programming language number one）。PL/I试图把ALGOL的块 结构、 FORTRAN的科学和数学计算功能以及 COBOL的记录和报表能力结合起来。但是，它 却远没有像 FORTRAN和COBOL那样流行。

尽管FORTRAN、ALGOL、COBOL和PL/I都有适用于家用计算机的版本，但是它们都不 具备BASIC所具备的那种对小计算机的影响力。

BASIC（beginner's all-purpose symbolic instruction code）是 Dartmouth数学系的 John Kemeny和Thomas Kurtz在1964年为 Dartmouth的分时系统开发的。Dartmouth的许多学生并非 主修数学或工程课程，所以他们不能在穿孔卡片和很难的程序设计语法上花费很多时间。Dartmouth的学生坐在终端前，只需在数字之后简单地敲入 BASIC语句，即可建立BASIC程序。数字表明程序中语句的顺序。没有数字在前的语句是对系统的命令，如 SAVE（存储 BASIC程 序到磁盘）、LIST（按顺序显示行）和 RUN（编译和执行程序）。第一批印刷的 BASIC指令手 册中的第一个 BASIC程序为：

```
10 LET X = ( 7 + 8 ) / 3
```

```
20 PRINT X
```

```
30 END
```

不同于ALGOL，BASIC不需要程序设计员来指定一个变量是按整数存储还是浮点数存储。不需要程序员操心，大多数数都是按浮点数存储。

许多后来的 BASIC版本是解释程序而不是编译程序的。前面讲过，编译程序是读取一个

源文件，并产生一个可执行文件；而解释程序读取源代码并在读的过程中直接执行而不生成可执行文件。解释程序比编译程序容易编写，但是，解释程序的执行时间却比编译程序的执行时间要慢。当比尔·盖茨（生于 1955年）和他的密友保罗·艾伦（生于 1953年）在 1975年 为Altair 8800编写BASIC解释程序并创立他们的公司——微软公司的时候， BASIC才开始应用到家用计算机中。

Pascal程序设计语言继承了 ALGOL的许多结构，但也包括了 COBOL的记录处理程序。该语言由瑞士计算机科学教授 Niklaus Wirth （生于 1934年）在 20世纪 60 年代后期设计而成。Pascal 在IBM PC程序设计员中很受欢迎，但却以一种特殊的形式——Turbo Pascal这种产品形式流行。该产品于 1983年由 Borland公司推出，售价为 \$49.95。Turbo Pascal （由丹麦学生 Anders Hejlsberg （生于 1960年）编写）是 Pascal的一个版本，提供了完整的集成化开发环境。

文本编辑器和编译程序集成在一个程序里，促进了快速编程。集成化开发环境在大型机上很流行，但 Turbo Pascal却首先在小机器上实现了。

Pascal对Ada也有很大影响。Ada是为美国国防部开发使用的一种语言，是以 Augusta Ada

Byron命名的。第 18章中已提到过这个人，他是查尔斯·巴贝芝的解析机的见证人。然后就有了 C语言，一种受到万般宠爱的程序设计语言。它于 1969年～1973年产生，大部

分是由贝尔电话实验室的 Dennis M.Ritchie 完成的。人们常常问为什么叫 C语言，简单的回答是它来自于一种早期的语言 B，B 是BCPL（Basic CPL）的一种简单版本，而 BCPL又来自于

CPL (combined programming language) 。

第22章曾提到过 UNIX操作系统被设计成可移植的形式。那时许多操作系统都是用汇编语言针对特定处理器而编写的。1973年，UNIX采用C来编写（更确切地说是重写）。从那时起，操作系统和C语言的关系就开始紧密起来。

C是很简洁的语言，例如，ALGOL和Pascal中用begin和end来定义的块，在C语言中用{ }来代替。下面是另一个例子，该例对程序设计员来说是很常见的，就是把一个常量与一个变量相加：

`i = i+5;` 在C语言中，可以简写为：`i+=5;`

如果只需要把变量加1（即增量），甚至可以这样来简写语句：

`i++;`

在16位或32位微处理器中，这样一条语句可以由一条机器码指令来实现。前面曾提到，许多高级语言不包括移位操作和按位逻辑操作，而这些是许多处理器所支

持的操作，C语言是个例外。另外，C语言的另一重要特点是支持指针，指针实质上是数字化的内存地址。由于C有许多操作类似于常见的处理器指令，因而有时候也把C语言归类于高级汇编语言。胜过于任何类ALGOL语言，C更接近于常用的处理器指令集。

然而，所有的类ALGOL语言——即指常用的程序设计语言，是在冯·诺依曼计算机体系结构基础上设计而成的。在设计计算机语言时，突破冯·诺依曼框架并不容易，而让人们来使用这种语言则更加困难。一个非冯·诺依曼的语言是LISP (LISt Processing)，是由John McCarthy在20世纪50年代末设计而成的，可用在人工智能领域。另一个与众不同且与LISP完全不同的语言是APL (A Programming Language)，是由Kenneth Iverson也在20世纪50年

代末 开发而成的。 **APL**采用了一个奇怪的符号集用来一次在整个数字数组上执行操作。

虽然类**ALGOL**语言仍保持着主导地位，最近几年，出现了叫作面向对象的程序设计语言， 使这类语言的地位得到加强。这些面向对象语言与图形化操作系统一起使用，图形化内容在下一章（即最后一章）将作介绍。

第 25 章 图形化革命

1945年9月10日,《Life》杂志的读者看到的大多是平常的一些文章和照片:有关第二次世界大战结束的故事,舞蹈家 Vaslav Nijinsky在维也纳生活的报道,一则有关美国汽车工人的图片报道。但那一期的杂志也有意想不到的东西:一篇 Vannevar Bush (1890-1974) 的关于科学研究的未来的展望性文章。Van Bush (人们这样称呼他) 在计算机历史上写下了重要的一笔。在 1927 年~1931 年任麻省理工学院工程教授期间,他设计了一种具有重大意义的模拟计算机—微分分析器。1945 年,在《Life》杂志发表这篇文章的时候,Bush 是科学研究和发展部的主管,负责协调美国在战争期间的科学活动,包括曼哈顿计划。

通过对两个月前第一次发表在《The Atlantic Monthly》上的那篇文章的精简,Bush 在

《Life》杂志上的文章《As We May Think》描述了未来一些假想发明,希望科学家和研究人员解决日益增多的技术杂志及文章。Bush 谈到了作为一种解决方案的微缩胶片,勾划出了一种他称之为 Memex 的设备来保存书、文章、记录和书中的图片。Memex 也可根据人们所想到的关系在这些东西之间建立有关某个主题的联系。他甚至设想出了一个新的职业群体,他们可在大量的信息载体之间牢固地建立起联系。

尽管描绘未来光辉前景的文章在 20 世纪很普遍,但《As We May Think》则不同,它描述的既不是关于减轻家务负担的设备的故

事，也不是关于未来交通运输或机器人的故事，而是关于信息及如何用新技术处理信息的故事。

从第一台继电器计算器制造出来已经历了 65 年，计算机变得越来越小、越来越快，也越来越便宜。这种趋势已改变了计算机的本质。当计算机越来越便宜时，每一个人都可拥有自己的计算机。计算机越小、越快，软件则变得越高级，同时机器可以完成更多的工作。

更好地利用这些额外功能和速度的一种方法就是改进计算机系统中至关重要的部分，即用户接口——人和计算机的交互点。人和计算机是差别很大的两种物质，但不幸的是，说服人们调整以适应计算机的特性是比其他方法更容易的方法。

早先，数字计算机根本上不是交互式的。有些使用开关和电缆编程，有些使用穿孔纸带或胶片编程。到 20 世纪 50 年代和 60 年代（甚至延续到 70 年代），计算机进化到使用批处理：程序和数据穿孔成卡片，然后读入到计算机内存，程序分析数据，得出一些结论，再在纸上打印出结果。

最早的交互式计算机使用的是电传打字机。如前一章讲到的 **Dartmouth** 分时操作系统（始于 20 世纪 60 年代早期）支持多个电传打字机，可以同时使用。在这样的系统里，用户在打字机上敲一行，计算机以回答一行或多行作为响应。打字机和计算机之间的信息交流全部是 **ASCII** 码流（或其他字符集），除了像回车换行这样简单的控制码外，差不多全是字符代码。事务只是按纸卷的方向进行。

然而，阴极射线管（在 70 年代就已经很普遍了）则没有这些限制。可以用软件以更灵活的方式来处理整个屏幕。然而，可能是为了设法保持显示器输出符合操作系统的显示输出逻辑，早先为小型计算机编写的软件不断地把 **CRT** 显示器作为“玻璃打字机”——一行一行地显

示直到布满整个屏幕，当有字符到达屏幕底端时，屏幕的内容要向上翻滚。**CP/M**和**MS-DOS** 中的实用程序都是以电传打字机的模式来使用视频显示器。也许原型电传打字机的操作系统是**UNIX**，它仍然保持着这种传统。

令人感兴趣的是，**ASCII**码字符集并不都适用于阴极射线管显示。在最初设计**ASCII**码时，代码 **1Bh**标识为 **Escape**，专门处理字符集的扩充。1979年，**ANSI**印发了一个标准，题为“使用**ASCII**码的附加控制”。该标准的目的是“为了适应可预见的有关二维字符图像设备输入输出控制的要求，包括有阴极射线管和打印机在内的交互终端...”

当然，**Escape**的代码 **1Bh**只有 1个字节，且只有一个含义。**Escape**通过作为可变长序列的开端来表达不同的功能。例如，以下这个序列：

1Bh 5Bh 32h 4Ah

即**Escape** 代码后面跟上字符 **[2J**，定义成删除整个屏幕并移动光标至左上角。这是在电传打字机上所不能实现的。下面这个序列：

1Bh 5Bh 35h 3Bh 32h 39h 48h

即**Escape**代码后面跟上字符 **[5; 29H**，把光标移到第 5行的第29列。

由键盘和 **CRT**组合而成，对来自远方计算机的 **ASCII**码（也可能是 **Escape**序列集合）作出响应，这样的设备有时称作哑终端。哑终端比打字机要快并且从某种意义上讲也更灵活，但是它并没有快到足以引起用户界面的真正创新。这种创新来自于 20世纪 70年代的小计算机，正如第21章中的假想计算机，这种计算机有视频显示存储器作为微处理器地址空间的一部分。

家用计算机显著区别于它们大而昂贵的伙伴的第一个标志可能是 **VisiCalc** 的使用。**VisiCalc**由**Dan Bricklin** (生于1951年)和**Bob Frankston**(生于1949年)设计和编程,于1979年推出,用于**Apple II**。**VisiCalc**在屏幕上呈现给用户一个二维电子数据表。在**VisiCalc**出现之前,报表通常是一张纸,使用行、列来进行一系列计算。**VisiCalc**用视频显示器取代了纸,使得用户可以移动报表,输入数据或公式,在进行修改后重新计算每一项。

令人吃惊的是 **VisiCalc**是不能复制到大型机上的应用程序。像**VisiCalc**这样的程序需要很快地刷新屏幕。因此,它直接向**Apple II**的视频显示器使用的随机访问存储器写入。该存储器是微处理器地址空间的一部分。大型分时计算机和哑终端之间的接口速度不是很快,从而使得电子报表程序不能使用。

计算机响应键盘、刷新视频显示器的速度越快,用户和计算机潜在的交互就越紧密。在**IBM PC**机出现的头10年(20世纪80年代),为它编写的大多数软件是直接写入显示存储器的。由于**IBM**建立了一套硬件标准,其他计算机厂商追随这一标准,使得软件厂商可以绕过操作系统直接使用硬件而不用担心他们的软件在某些机器上不能正确运行(或根本不能运行)。如果所有的**PC**“克隆体”都与它们的视频显示器有不同的硬件接口,则对软件厂商来说要满足所有不同的硬件设计是非常困难的。

IBM PC所使用的早期的应用程序大多数只有字符输出而没有图形输出。使用字符输出同样能使得应用程序的执行速度加快。如果视频显示器设计得如第21章所描述的那样,则程序只需简单地把某个字符的**ASCII**码写入内存就可以在屏幕上显示出该字符。使用图形视频显示的程序需要写入8个或更多的字节到内存中才能画出文本字符的图形。

从字符显示到图形显示的变化是计算机革命中极其重要的一步,然而图形方式下计算机硬件和软件的发展比文本和数字方式下计算机硬件和软件的发展要慢的多。早在1945年,

冯·诺依曼就设想了一种像示波器一样的显示器，可用来使信息图形化。但是，直到 20 世纪 50 年代早期，计算机图形才开始成为现实。当时麻省理工学院（得到 IBM 资助）建立了林肯实验室来开发计算机，用于美国空军的空中防卫系统。该项目称为 SAGE（semi-automatic- ground environment），有一个图形显示屏帮助操作员分析大量数据。

在 SAGE 这样的系统中使用的早期视频显示器不像今天我们在 PC 机中所用的显示器。今天普通的 PC 机显示器是光栅显示器。就像电视机，所有的图像由一系列水平光栅线组成，由一个电子枪射击光束很快地前后移动扫过整个屏幕形成光栅。屏幕可以看成是一个大的矩形点阵，这些点称为像素。在计算机里，一块内存专门供视频显示使用，屏幕上的每一个像素由 1 位或多位表示。这些位值决定像素是否亮，是什么颜色。

例如，今天多数计算机显示器有至少水平方向 640 像素的分辨率，垂直方向 480 像素的分辨率，像素的总数是这两个数的乘积 307 200。如果 1 个像素只占用 1 位内存，则每个像素只局限于两种颜色，即通常的黑、白色。如，0 像素为黑，1 像素为白。这样的视频显示器需要 307 200 位的内存，即 38 400 字节。

随着可能的颜色数目的增多，每个像素需要更多的位，显示适配器也需要更多的内存。例如，每个像素可以用一个字节来编码灰度。按照这样的安排，字节 00h 为黑，FFh 为白，之间的值代表不同的灰度。

CRT 上的彩色由三个电子枪产生，每一个分别负责三原色红、绿、蓝中的一种（可以用放大镜来观察电视机或彩色计算机屏幕以验证它的正确性。由不同的原色组合来显示图像），红色和蓝色的组合是黄色，红色与绿色的组合是洋红色，蓝色和绿色的组合是青色，三原色的组合是白色。

最简单的彩色图像显示适配器每个像素需要 3 位。像素可以如以下这样编码，每一个原色 对应1位：

位	颜色
000	黑
001	蓝
010	绿
011	青
100	红
101	洋红
110	黄
111	白

但是，这种方式只适合于简单的类似卡通画的图像。许多现实世界中的颜色都是红、绿、蓝按不同级别组合而成的。如果用 2 个字节来表示一个像素，则每一个原色可分配 5 位（1 位保留），这样可以给出红、绿、蓝三种颜色各 32 种不同的级别，即总共可有 32 768 种不同的颜色。这种模式通常称作高彩色或千种颜色。

下一步是用 3 个字节来表示一个像素，每种颜色占一个字节。这种编码模式使红、绿、蓝

三种颜色各有 256 种不同的级别，这样总共有 16 777 216 种不同的颜色，通常称作全彩色或百万种颜色。如果视频显示器的分辨率为水平 640 像素，垂直 480 像素，则总共需要 921 600 字节的存储容量，即将近 1M 字节。

每个像素所占的位数有时也称作颜色深度或颜色分辨率。不同颜色数量与每个像素所占的位数的关系如下：

$$\text{颜色数} = 2^{\text{每个像素所占位数}}$$

视频适配卡只有一定数量的存储器，这样它所能达到的分辨率和颜色深度将受到限制。例如，一个具有 1M 字节存储器的视频适配卡可以达到 640×480 的分辨率，每个像素占 3 个字节。如果想用 800×600 的分辨率，则没有足够的存储器给每个像素分配 3 个字节，而要用 2 个字节来表示一个像素。

尽管现在使用光栅显示器似乎是理所当然的，但在早期由于需要大量存储器，使用光栅显示器就不太实际。SAGE 视频显示器是矢量显示器，比电视机更像示波器。电子枪可以定位到显示器上任何部分的点，并且直接画出直线或曲线。利用屏幕上图像的可持续性使得能用这些直线和曲线来形成最基本的画面。

SAGE 计算机也支持光笔，操作者可用在显示器上改变图像。光笔是特殊的设备，看起来像一只铁笔，一端连有电线。运行适当的软件后，计算机可以检测到光笔指向的屏幕位置，并随着光笔的移动而改变图像。

光笔是如何工作的呢？即使是技术专家，在第一次看到光笔的时候也会迷惑不解。关键在于光笔不发射光——它检测光。在 CRT 中（无论是用光栅还是向量显示），控制电子枪移动的电路也可以确定从电子枪射出的光何时打到光笔上，从而确定光笔正指向屏幕的什么位置。Van Sutherland(生于 1938 年)是预见到新的交互式计算时代的多个人之一，他在 1963 年

示范了一个他为 SAGE 计算机开发的名为 Sketchpad（画板）的具有革命性意义的图形程序。画板可以存放图像信息到存储器，并且可以把图像显示在屏幕上。另外，还可以用光笔在显示器上画图并修改，同时计算机会一直跟踪它。

另外一个交互式计算的设想家是 Douglas Engelbart(生于 1925 年)。他曾读过 1945 年 Vannevar Bush 发表的文章《As We May

Think 》，并在 5 年后开始用一生的时间致力于研究新的计算机界面。20 世纪 60 年代中期，当他在 Sanford 研究院时，他彻底重新考虑了输入设备，提出了用有五个尖端的键盘 (此设备没有流行) 以及一个他叫作鼠标的有轮子和按钮的小设备来输入命令。鼠标现在已广泛用来移动屏幕上的指针，以选择屏幕上的对象。

恰逢光栅显示器在经济上切实可行，许多早期的热衷于交互式图形计算的人们（尽管没有 Engelbart）也已聚集在 Xerox 公司里。Xerox 公司于 1970 年建立了 Palo Alto 研究中心

(PARC)，其中一项就是资助开发产品，以使得公司能进入计算机界。也许 PARC 最著名的设想家是 Alan Kay（生于 1940 年），当他 14 岁的时候碰巧看到了 Robert Heidein 的小故事里描述的 Van Bush 的微缩胶片图书馆，并已设想了一种他称为 Dynabook 的轻便计算机。

PARC 的第一个大工程是 Alto，于 1972 年～1973 年期间设计和制造出来。按照那个年代的标准，这是一个给人深刻印象的产品。它是安装在地板上的系统单元，有 16 位处理器、2 个 3MB 的磁盘驱动器、128KB 内存（可以扩充到 512KB），以及一个三个按钮的鼠标。在 16 位单片微处理器之前出现，Alto 的处理器由大约 200 个集成电路组成。

Alto 的视频显示器是它与众不同的地方之一。屏幕的大小和形状与一张纸差不多——8 英寸宽 10 英寸高。它以光栅图形模式工作，有 606 个水平像素和 808 个垂直像素，这样总共 489

648 个像素。每个像素占 1 位存储器，即每个像素不是黑色就是白色。用于视频显示的存储器容量是 64KB，占用部分处理器的地址空间。

通过写入到视频显示存储器，软件可以在屏幕上画出图画或显示不同字体和大小的文本。通过在桌上滚动鼠标，用户可以在屏幕上定位指针，与屏幕上的对象进行交互。视频显示器

并非像电传打字机那样按行显示用户输入，按行写出程序输出。视频显示器的屏幕是一个二维的、高密度的信息阵列和更直接的用户输入源。

在20世纪70年代末期，为 Alto所写的程序显示出了许多令人感兴趣的特点。多个程序可放到窗口中并同时显示在屏幕上。Alto的视频图像使得软件超出文本范畴，并真正反映用户的思想。图形对象（如：按钮、菜单及称作图标的小图画）成为用户接口的一部分。鼠标用来选择窗口或触发图形对象，执行程序功能。

这就是软件，不仅提供了用户接口而且已达到与用户亲密交互的程度。软件进一步扩展了计算机的应用领域而不再仅进行简单的数字操作。软件设计出来就是——引用Douglas Engelbart 在1963年写的论文的标题——《for the Augmentation of Man's Intellect》。

在Alto上开发的 PARC只是图形用户界面即 GUI（graphic user interface）的开始。但 Xerox 公司并没有销售 Alto（如果要销售的话，价格将在3万美元以上）。经过10多年，高售价的思想体现在一种成功的消费品上。

1979年,Steve Jobs 和来自苹果计算机公司的小组参观了 PARC，看到的東西给他们留下了深刻的印象。但是，他们花费了三年多的时间才推出了具有图形接口的计算机，这就是1983年1月推出的不太受欢迎的 Apple Lisa。而在1年以后，苹果公司就推出了很成功的 Macintosh。初始的 Macintosh配备有 Motorola 6800 微处理器、64KB的ROM、128KB的RAM、一个3.5英寸的磁盘驱动器（可以存储400KB）、一个键盘、一个鼠标和一个视频显示器。该视频显示器水平512个像素，垂直342个像素（CRT测量对角只有9英寸），总共175 104个像素。每一

个像素用1位内存，颜色为黑、白，这样视频显示 RAM大约需要22KB。

最初的 **Macintosh** 硬件做得很好，但是很难进行变革。1984年，出现了使 **Mac** 与其他计算机完全不同的 **Macintosh** 操作系统，那时通常称作系统软件，后来称为 **Mac OS**。

像 **CP/M** 或 **MS-DOS** 这样的基于字符的单用户操作系统不是很大并且没有扩展的应用程序接口（**API**）。正如第 22 章解释的那样，这些基于字符的操作系统所需要的是使用文件系统的程序。而像 **Mac OS** 这样的图形操作系统则非常大且有几百个 **API** 函数，每一个都由一个描述该函数功能的名称来标识。

像 **MS-DOS** 这样的基于字符的操作系统只需提供几个简单的 **API** 函数就能使得应用程序在屏幕上以电传打字机方式显示字符，但 **Mac OS** 这样的图形操作系统必须提供一种方法才能使得程序可以在屏幕上显示图形。从理论上讲，可以通过实现一个 **API** 函数来完成，该函数使得应用程序可以设置某个水平和垂直坐标的像素的颜色。但是，实际证明这种方法效率不高且图形显示的速度很慢。

因而由操作系统提供完整的图形编程系统就显得非常有意义，这意味着操作系统需包含有画线、画矩形、画椭圆（包括圆）和字符的 **API** 函数。线可以由实线或虚线或点组成；矩形和椭圆可以用不同的填充模式来填充；字符可以显示成不同字体和大小并具有不同效果，如：黑体和下划线等。图形系统负责确定如何在显示器上把这些图形对象作为点阵来显示。

图形操作系统下运行的程序使用相同的 **API** 在计算机视频显示器和打印机上画出图形。因此，字处理应用程序在屏幕上显示的文档可以与打印出来的文档非常相似。这种特点称为 **WYSIWYG**，是“**What you see is what you get**（所见即所得）”的英文缩写。这是喜剧演员 **Flip Wilson** 在 **Geraldine** 角色中所贡献的计算机行话。

图形用户界面的吸引力部分体现在不同应用程序的工作大致相同，且与用户体验关系不

大。这意味着操作系统还必须支持 API函数从而使得应用程序可以实现用户界面的不同部分，如按钮和菜单等。GUI通常看起来是友好的用户环境，它对编程人员来说同样是很重要的环境。编程人员在原来的基础上就可以实现现代用户界面。

在Macintosh推出之前，几家公司就已开始创建用于 IBM PC 及其兼容机的图形操作系统。从某种意义上讲，Apple开发人员的工作要容易一些，因为他们硬件和软件一起设计。Macintosh系统软件只支持一种类型的磁盘驱动器，一种类型的视频显示器和两种打印机。但在PC上实现的图形操作系统需要支持许多不同的硬件。

另外，尽管 IBM PC 很早（1981年）就已推出了，但多数人已习惯于用他们喜好的 MS-DOS应用程序且没有准备放弃它们。PC机的图形操作系统需要考虑的一个重要方面是，要使得MS-DOS应用程序的运行就像是专门为新的操作系统设计的应用程序一样。（Macintosh上就根本不能运行 Apple II软件，因为它采用了不同的微处理器。）

1985年，Digital Research 公司（CP/M的后续公司）推出了 GEM（图形环境管理器）；Visicorp公司(销售Visilalc的公司)推出了 VisiOn；Microsoft 公司发布了 Windows 1.0版，它很快被认为是“视窗战争”中最有可能的胜利者。然而，直到 1990年3月发布 Windows 3.0，Windows才开始吸引大量的用户。从那时起，它的普及率不断提高。到今天，已有大约 90%的 微机上使用的操作系统是 Windows。Macintosh和Windows除了具有相同的外在表现外，它们的API是非常不同的。

理论上讲，除了图形显示外，图形操作系统并不比字符操作系统需要更多的硬件，甚至不需要硬盘驱动器。最初的 Macintosh 没有，Windows 1.0也不需要。Windows 1.0甚至不需要鼠标，尽管每个人都认为用鼠标操作更容易一些。

然而(这儿一点也不奇怪)，随着微处理器越来越快，内存和外存越来越大，图形用户界面也越来越流行。越来越多的特点增加到图形操作系统，至使它们越来越大。今天的图形操作系统通常需要 200MB的硬盘空间和 32MB以上的内存。

图形操作系统的应用程序几乎没有是用汇编语言编写的。早期 Macintosh上应用程序的流行语言是 Pascal。对于 Windows应用程序来说，流行语言是 C。但PARC再次使用了一种不同的方法。大约从 1972年开始，PARC的研究人员就在开发一种称为 Smalltalk的语言，体现了面向对象程序设计，即 OOP的概念。

通常，高级程序设计语言的代码（通常以 set、for、if 这样的关键字开头的语句）和数据

（用变量来表示的数）之间有区别。毫无疑问，这种区别源自冯·诺依曼计算机体系结构。在这种体系结构里，要么是机器码，要么是机器码用于操作的数据。

而在面向对象的程序设计中，对象是代码和数据的组合。在对象中，数据存储的实际方法只能通过与该对象相关联的代码才能理解。对象通过发送或接收消息来与其他对象通信，它给一个对象发送指令或从那里获得信息。

面向对象语言通常有助于编写用于图形操作系统的应用程序，因为编程人员可以用与用户感知对象的同样的方式来处理屏幕上的对象（如：窗口和按钮等）。在面向对象语言中，按钮是对象的一个例子。屏幕上的按钮有一定的尺寸和位置，并显示一些文本或小的图画，所有这些是与对象相关的数据。与对象关联的代码确定用户何时用键盘或鼠标按下按钮，并且发送一个标明该按钮被触发的消息。

然而，最流行的微机上的面向对象语言是传统的类 ALGOL语言的扩展，如 C和Pascal。最

流行的由 C 扩展的面向对象语言是 C++（前面讲过，两个 + 是增量操作）。C++ 大部分是由贝尔实验室的 Bjarne Stroustrup（生于 1950 年）完成的，开始作为转换程序，用来把用 C++ 编写的程序转换成 C 程序（尽管 C 程序很难看也很难读），C 程序可以像通常一样编译。

当然，面向对象语言并不能比传统语言多做些什么。但是编程是解决问题的方式，而面向对象语言使得编程人员能够考虑那些在结构上通常更好的不同的解决方法。也可以——尽管不是那么容易——用面向对象语言编写程序，编译后可在 Macintosh 上或 Windows 下运行。这样的程序并不直接涉及到 API 而是使用称作 API 函数的对象。两个不同的对象定义用来编译用于 Macintosh 或 Windows API 的程序。

许多小型机上的编程人员不再用命令行编译程序。取而代之的是编程人员开始采用集成开发环境（IDE），即在一个方便的程序里集成有所需的所有工具并且该程序可像其他图形应用程序一样运行。编程人员还利用一种称作可视化编程的技术，通过鼠标汇集按钮及其他组件来设计交互窗口。

第 22 章中讲到了文本文件。这种文件只包含有 ASCII 字符，方便人们阅读。在使用基于字符的操作系统时，文本文件是在应用程序之间交换信息的理想工具。文本文件的一个最大优点就是它们是可检索的——即程序可以查看许多文本文件并确定它们中的哪一个包含有某一字符串。但是，一旦某个操作系统中有一个工具可用来显示不同字体、不同大小及不同效果

（如斜体、黑体和下划线），则文本文件似乎就很不适用了。其实，许多字处理程序以独有的二进制格式来存储文档。文本文件同样也不适用于图形信息。

但是，可以同文本一起编码信息（如字体定义及段落编排），且仍然得到可阅读的文本文件。关键是选用一个转换字符来表示这

些信息。在 **Microsoft**设计的 **RTF** (**rich text format**) 中, 作为在应用程序之间交换格式化文本的一种方法, 花括号 {} 及反斜杠 \ 用来封装信息, 标明文本采用何种格式。

PostScript是把这种概念发挥到极致的一种文本格式。**PostScript**由 **Adobe**系统的创始人之一 **John Warnock** (生于 1940年) 设计。这是一种通用的图形编程语言, 主要用来在高端计算机的打印机上画出字符或图形。

把图形结合到个人计算环境是越来越好、越来越便宜的硬件的直接结果。微处理器越来越快, 存储器越来越便宜, 视频显示器及打印机分辨率不断增加且具有更多种颜色, 所有这些促进了计算机图形的使用。

计算机图形产生于两种不同方式, 与早些时候为区分图形视频显示器所用的词一样: 矢量和光栅。

矢量图形用直线、曲线及填充的域来生成图形, 这是计算机辅助设计 (或 **CAD**) 程序的领域。矢量图形在工程和结构设计中具有重要用途。矢量图形可以按元文件的格式存放到文件中。元文件是矢量图形制作命令的聚合, 这些命令通常以二进制形式编码。

矢量图形采用直线、曲线及填充的域, 因而非常适合于桥梁设计等, 但不能指望它来实际显示建造的桥梁的效果。桥梁效果图是现实世界的图像, 用矢量图形来表示太复杂, 因而很难表示出来。

光栅 (也称作位图) 可用来解决这一问题。位图把图像编码成位的矩形阵列, 该阵列对应于输出设备上的像素。就像视频显示器一样, 位图具有空间度 (或分辨率), 即指图像按像素表示的宽度和高度。位图也有颜色度 (或颜色分辨率/颜色深度), 是指每一个像素对应的

位数。位图中的每一个像素用相同的位数来表示。尽管位图图像是二维的，但位图本身只是一个字节流——通常从最上面一行像素开始，接

着是第2行、第3行等等。一些位图图像是使用为图形操作系统设计的画笔程序通过“手工”生成的，还有一些位

图图像由计算机代码来生成。现在，位图经常用来表现现实世界的图像（如：照片），有几种不同的硬件可用来把现实世界的图像输入到计算机中，这些设备通常用到称作电荷耦合

（CCD）的器件，它是一种半导体器件，在光照下会放电。一个 CCD单元用来采样像素。扫描仪是这些设备中最古老的一种。就像影印机一样，它用一行 CCD扫过印制图像（如：

照片）的表面。随着光的强度不同，CCD具有不同的电荷积累。与扫描仪一起工作的软件把图像转换成位图存放在文件里。

视频摄像机用二维 CCD单元阵列来捕捉图像。通常这些图像录制在录像磁带上。但视频输出也可以直接送到视频帧输入器，这是用来把模拟信号转换成像素值阵列的一块板。帧输入器可以使用任何普通的视频源，如 VCR或激光影碟机，甚至直接来自于有线电视盒。

最近，数字照相机价格已适合家庭购买，它看起来很像一般的照相机。但是，数字相机不用胶片，而是用 CCD阵列来捕捉图像并直接存储到照相机的存储器中，然后传输到计算机中。

图形操作系统常常支持某种格式的位图文件的存储。Macintosh使用Paint格式，这个命名参考了创立这种格式的 MacPaint程序。（Macintosh的PICT格式综合了位图和矢量图形，是它的首选格式。）Windows里的格式是 BMP，它是位图文件的扩展名。

位图可能很大，采用一些方法压缩它们是很有用处的。这些工作都归入到计算机科学中 称为数据压缩 的范畴。

假设正在处理图像，如前所述，每个像素占 3 位。若有一个图片有天空、一栋房子和一块 草坪，这样的图片可能就会有大量的蓝色和绿色。也许，位图的最上面一行是 72 个蓝色像素，如果有一些方法能够在文件中编码这 72 个数字，以表示蓝色像素重复 72 次，则位图文件可能会变得更小。这样的压缩称为行程编码，即 RLE（run-length encoding）。

一般办公用的传真机采用 RLE 压缩方法，在通过电话线传送之前压缩图像。由于传真机 只把图像看成黑色和白色而没有灰度和彩色，因此，通常有很长串的白色像素。

已经流行了 10 多年的位图文件格式是图形交换格式即 GIF，由 Compu Serve 公司于 1987 年 开发出的。GIF 文件采用称为 LZW 的压缩技术，“LZW”代表它的创建者：Lemplel、Ziv 和 Welch。LZW 比 RLE 效果更好，因为它检测不同像素值的模式而不仅仅是针对具有相同值的像素的连续串。

RLE 和 LZW 都是无损失的压缩技术，因为从压缩数据中可以重新生成完整的初始文件。换句话说，压缩是可逆的。很容易证明可逆的压缩方法并不适用于所有类型的文件。在某些 情况下，“压缩”文件比初始文件还要大。

最近几年，有损失的压缩技术很流行。有损失的压缩是不可逆的，因为某些初始数据被 丢弃了。不要用有损失的压缩技术来压缩电子报表或文字处理文档，因为每一个数字或文字 也许都是很重要的。但这并不妨碍用有损失的压缩技术来压缩图像，因为只要丢弃的数据不 会使得图片的整体效果有太大差别即可。这就是有损失的压缩技术用于可视心理研究的原因， 它可以研究人的视觉，以确定什么重要，什么不重要。

最重要的用于位图的有损失的压缩技术统称为 JPEG。JPEG表示联合图像专家组（joint photography experts group），它描述了几种压缩技术，一些是无损失的，一些是有损失的。

把元文件转换成位图文件很简单。因为视频显示存储器与位图在概念上是一致的。如果一个程序知道如何在视频显示存储器中画一个元文件，则它也知道如何在位图上画元文件。

但是，把位图文件转换成元文件就没那么容易，有些复杂的图像甚至不能转换。与这项工作相关的一项是光学字符识别，即 OCR（optical character recognition）。如果一个位图上有一些字符（从复印机来的，或扫描页面得到的）并且需要转换成 ASCII 码，就可用 OCR。OCR 软件需要分析位的模式并确定它们代表什么字符。由于这项工作的算法很复杂，OCR 软件并不是 100% 准确。即使有些不准确，OCR 软件也试图把手写体转换成 ASCII 码字符。

位图和元文件都是用数字表示的可视信息。音频信息也可以转换成位和字节。

1983年，随着激光唱机的出现，数字化音响激起了消费者的热情，它也成为了最大的电子消费品。CD由Philips和Sony公司开发，在一个直径 12cm 的盘上一面可存储 74 分钟的数字化声音。之所以选择 74 分钟是因为贝多芬的第九交响曲刚好可以放在一张 CD 上。

CD 上的声音编码采用脉冲编码调制技术，即 PCM (pulse code modulation)。不管它的名字多么奇怪，从概念上讲，PCM 是很简单的处理过程。

声音是振动产生的。人们的声音是振动，大号的声音是振动，森林里树倒下的声音也是振动，它们使得空气分子移动，空气一会儿挤压一会儿弹开，一会儿压缩一会儿放松，一会儿向后一会儿

向前，每秒钟进行着成百上千次运动。空气最终震动耳膜，使得我们能够听到声音。声波可以用 1877年爱迪生的第一台电唱机上用来录制和播放背景音乐的锡箔圆桶表面上

的凸起和凹陷来模拟。直到出现 CD之前，这种录制技术也很少改变，虽然圆桶换成了盘片，锡箔换成了塑性材料即塑料。早期的电唱机是全机械的，后来使用电子放大器来放大声音。麦克风上的可变电阻把声音转换成电流，喇叭中的电磁铁把电流转换回声音。

代表声音的电流并非本书中所讲的 1/0数字信号。声波是连续变化的，而产生这种电流的电压也是如此。电流是声波的模拟。一种称作 模拟数字转换器（ADC）的器件——通常在一个芯片上实现——把模拟电压转换成二进制数。ADC的输出是若干位数字信号——通常为 8、12或16位——用来表明电压的相对级别。例如，12位 ADC把电压转换成 000h~FFFh之间的数，从而区分 4096个不同的电压级别。

在脉冲编码调制这种技术里，代表声波的电压按照恒定的速率转换成数值。这些数以小孔的形式刻在光盘表面，从而存储在 CD上。通过从 CD 表面反射的激光可以读出这些信息。在播放的时候，这些数又通过数字/模拟转换器即DAC转换成电流。（DAC也用在彩色图形板上，用来把像素值转换成模拟的彩色信号送到显示器。）

声波电压以恒定的速率转换成数字，该速率称为采样速率。1928年，贝尔实验室的 Harry Nyquist证明了采样速率至少为需要记录和播放的信号的最大频率的两倍。通常认为人们能听到的声音的频率范围为 20~20 000 赫兹。CD所用的采样频率比最大频率的两倍还要大一些，定义为每秒采样 44 100次。

每个样本所用的位数取决于 CD的动态范围，即记录和播放的声音的最高频率与最低频率之差。这有些复杂：电流不断地变化来模拟声波，尖峰称为声波的振幅。我们所感受到的声音强度是振幅的两倍。1贝尔（bel）表示强度的 10倍增强；1分贝(decibel)是1贝尔的 1/10，表示人们所能感受的声音的几乎最小的强度变化。

每个样本用 16位表示，这样可以表示 96分贝的动态范围，差不多是从能听到的声音的阈值（低于这一值则不能听见）到能忍受却不感到痛苦的声音的阈值的差。CD盘中用 16位表示一个样本。

所以，CD盘中每秒声音有 44 100个采样样本，每个样本 2个字节。立体声则需要两倍的采样信息即每秒总共 176 400字节，每分 10 584 000字节。（现在可以知道为什么在 20世纪80年代

之前声音的数字记录不是很普遍。）CD上74分钟的立体声需要 783 216 000字节。数字化声音与模拟声音相比具有很多众所周知的优点。特别是，无论何时复制模拟声音

（例如从录音磁带生成电唱片）都会有一些失真。而数字化声音是数字信息，总可以如实地转录和复制。过去常常是电话信号传输线路越长则声音越糟。现不再是这样了，因为现在许多电话系统都是数字的，跨越一个国家的呼叫信号就像跨越一条街道一样清晰。

CD也可像存储声音一样来存储数据。用得最广泛的用来存放数据的CD称作 CD-ROM

（CD只读存储器），通常 CD-ROM最多可存储约 660MB。今天，许多计算机中都装有 CD 驱动器，许多应用程序和游戏都在 CD-ROM中。

大约10年前，声音、音乐、视频开始进入个人计算机中，这称为多媒体。现在多媒体已经很普遍了，也不需要特别的名称。今天出售的许多家用计算机有声卡，内含一个 ADC用来把从麦克风来的声音录制成数字，还有一个 DAC用来通过喇叭播放录制的声音。声音可以以波形文件存放在磁盘中。

因为在家用计算机中录制和播放声音并不总是需要达到 CD 的质量，所以 Macintosh和 Windows提供低的采样速率，如 22 050、11

025和8000赫兹，以及较小的 8位样本信息和单频 度录制。声音以每秒 8000字节来录制，即每分 480 000字节。

人们从科幻电影和电视中知道，未来的计算机可以用英语与用户交谈。一旦计算机有了 数字化录制和播放声音的硬件，则所有通向这一目标的其他工作就可用软件来完成。

使计算机能讲人们能识别的单词和句子的方法有两种。一种方法是让人们录制句子段落、短语、单词及数字，然后存储在文件中，并且用不同的方法串在一起。这种方法通常用在通过电话访问的信息系统中，它在只需播放有限的单词和数字组合的情况下能很好地工作。

一种常见的声音合成形式涉及到一个用来把 ASCII码字符转换成波形数据的进程。例如，由于英语拼写并不总是一致的，所以这样的软件系统用一个词典或复杂算法来确定单词的确切发音。基本的音节（称作音素）组合成整个单词。通常软件需要做一些调整，例如，如果一个句子后面跟着问号，则最后一个单词的声音频率必须增加。

声音识别——把波形数据转换成 ASCII码字符——是一个更复杂的问题。的确，许多人在理解口语的方言方面有一些问题。在个人计算中使用听写软件时，通常需要训练以便能合理转录某个人所说的话。其中涉及的一个问题已超出了转换成 ASCII码文本的范围，即编程使计算机“理解”所说的话。这个问题是人工智能的研究领域。

今天，计算机中的声卡也提供小的电子音乐合成器，它能模仿 128种不同的音乐乐器和 47种不同的打击乐器，称作 MIDI合成器。MIDI即乐器数字接口，在 20世纪80年代早期由电子音乐合成器制造者协会开发出来，用来把这些电子乐器互相连接起来并连到计算机中。

不同种类的 MIDI 合成器用不同的方法来合成乐器的声音，其中一些比另一些更逼真。MIDI合成器的性质已远远超过了 MIDI定义的范畴。所要做的无非是通过演奏声音来响应短消息——通常长度为 1、2或3字节。MIDI消息常常指明需要什么乐器、将要演奏哪个音符，或正

在演奏的音乐要停止演奏。

MIDI文件是加上时间信息的 **MIDI**消息的集合。通常，一个 **MIDI**文件包含有计算机上的 **MIDI**合成器所能演奏的所有音乐成分。要包含同样的音乐，**MIDI**文件通常比波形文件小得多。按照相对大小来说，如果说一个波形文件像位图文件，则 **MIDI**文件就像矢量图形元文件。**MIDI**文件的不足之处在于：以 **MIDI**文件编码的音乐可能在一个 **MIDI**合成器上演奏得很好，但在另一个合成器上演奏出来却很糟。

多媒体的另一个特征是数字化电影。电影和电视图像的移动效果可以通过快速显示一系列静止图像来达到。这些单个图像称为帧。电影以每秒 24 帧的速率来播放，北美电视每秒为 30 帧，世界上其他许多地方的电视每秒为 25 帧。

计算机中的电影文件由一系列有声音的位图简单组成。但如果不经压缩，一个电影文件将包含大量的数据。例如，假设一个电影每一帧的大小是 640×480 像素的计算机屏幕，有 24 位彩色，则每帧有 921 600 字节。按每秒 30 帧，则每秒 27 648 000 字节。一直乘下去，则每

分钟为 1 658 880 000 字节，一个两小时的电影有 199 065 600 000 字节，大约 200GB。这就是为什么许多在个人计算机上播放的电影又小又短又跳跃的原因。

JEPG压缩方法用来减少存放静止图像所需的数据量，而 **MPEG**压缩方法用于存放运动图像。**MPEG**代表移动图像专家小组。移动图像压缩技术利用的是这一事实，即某一帧通常包含从前一帧复制来的大量信息。

对不同的媒体来说，有不同的 **MPEG**标准。**MPEG-2**用于高清晰度电视（**HDTV**）及数字视盘（**DVD**），也叫数字万用盘。**DVD**的大小与 **CD**一样，但可以两面记录且每一面有两层。在

DVD中，视频信息按照大约 50倍这样的因子进行压缩，所以，一个两小时的电影只需 4GB，且只需放在一面的一层。如果用两面和两层，则 DVD的容量可达到大约 16GB，约是 CD容量的25倍。可以预见，DVD最终将取代 CD-ROM来存储软件。

CD-ROM和DVD-ROM是不是 Vannevar Bush的预言在今天的实现？他开始设想的 Memex 是用缩微胶片，但用 CD-ROM和DVD-ROM更适合。电子媒体比物理媒体具有优越性，因为前者更容易检索。遗憾的是，很少有人同时访问多个 CD或DVD驱动器。我们所接触的 Bush概念中的文件柜并不涉及存储桌面上所需的所有信息，它涉及的是互连计算机使得它们共享信息并更有效地利用存储空间。

公开从远程操作计算机的第一人是 George Stibitz，正是他在 1930年设计了贝尔实验室的继电器计算机。继电器计算机的远程操作于 1940年在 Dartmouth进行了演示。

电话系统是用来在线路上传输声音的，而不是位。电话线路上传输位需要将位转换成声音然后再转换回位。一种频率和一种振幅的连续声波（称作载波）并不能传送真实的信息。但是，如果改变声波的一些东西——换句话说，在两种不同的状态之间调制声波——则可以表示0和1。在位和声波之间的转换由称作调制解调器的设备来实现。调制解调器是串行接口的一种形式，因为一个字节有位是一个接一个传输的，而不是同时传输的。（打印机通常通过并行接口与计算机连接：8根线同时传输一个字节。）

早先的调制解调器采用称作频移键控（FSK）的技术。以 300bps传输的调制解调器把 0调制到1070赫兹，把 1调制到1270赫兹。每个字节以一个起始位开始，以一个停止位结束，所以每个字节需要 10位。以 300bps的速率传输，每秒只传输 30个字节。许多现代调制解调器用更高级的技术能达到超过 100倍的速率。

早期家用计算机爱好者可以用计算机和调制解调器建立公告牌系统（**BBS**），其他计算机可以接入并下载文件，即从远程计算机传输文件到自己的计算机。这种概念扩展到了如 **CompuServe** 这样的大型信息服务。在大多数情形中，通信完全采用 **ASCII** 码字符形式。

Internet 则不同于这些早期的成就，因为它是分散的系统。**Internet** 其实就是计算机之间相互通信的协议集合，其中最主要也是最重要的是 **TCP/IP**，由传输控制协议（**TCP**）和网际协议（**IP**）组成。与通过线路只传输 **ASCII** 码字符不同，**TCP/IP** 的发送程序把大的数据块分割成小的包，在传输线路（通常是电话线）上独立传输，在另一端重新装配。

Internet 上流行的图形部分是 **World Wide Web**，采用 **HTTP**，即超文本传输协议。在 **Web** 页面上看到的数据由称作 **HTML** 即超文本标记语言的格式来定义。这些名词中超文本这个词用来描述相关信息的链接，非常类似于 **Vannevar Bush** 提到的 **Memex**。一个 **HTML** 文件可以包含到其他 **Web** 页面的链接，从而容易地访问它们。

HTML 与前面讲到的富文本格式（**RTF**）很相似，都包含有带有格式信息的 **ASCII** 码文本。**HTML** 也可包含 **GIF** 文件、**PNG**（portable network graphics）文件和 **JFIF**（**JPEG** 文件交换格式）文件等格式的图形。许多 **World Wide Web** 浏览器可以浏览 **HTML** 文件，这是文本格式的一个优点。把 **HTML** 文件定义成文本文件的另一个优点是它更容易查找。不管它的名称如何，

HTML 并不是像我们在第 19 章和第 24 章讲到的那些真正的程序设计语言。**Web** 浏览器读取

HTML 文件并依照它来编排文本和图形格式。

当你在浏览某个 **Web** 页面并在上面操作时执行一些特殊的程序代码是有用的，这些代码 可以在服务器（指那些存储初始 **Web** 页面的计算机）或客户机上运行，客户机即自己的计算机。在服务器端，通常所要做的全部工作（例如对客户端填写的在线表格的解释）可以通过 公共网关接口（**CGI**）脚本来处理。在客户端，**HTML** 文件可以包含简单的程序设计语言，如 **Java Script**。Web 浏览器就像解释 **HTML** 文本一样来解释 **Java Script** 语句。

为什么一个 **Web** 站点不能简单地提供一个可以在你的计算机上执行的程序呢？这涉及到 一个问题，你的计算机是什么？如果是 **Macintosh**，则需要一个包含 **PowerPC** 机器码的可执行文件并使用 **Mac OS API**；**PC** 兼容机需要一个包含 **Intel Pentium** 机器码的可执行文件，并使用 **Windows API**。但还有其他计算机及图形操作系统。而且，你也不想不加选择地下载可执行文件，它们可能来自于不值得信赖的地方且带有某种恶意。

对这些问题的回答可由 **Sun** 公司的 **Java** 语言来提供（不要与 **JavaScript** 混淆）。**Java** 是一个 完美的面向对象的程序设计语言，非常像 **C++**。前面几章里已经解释了编译语言（产生包含 机器码的可执行文件）和解释语言（不产生可执行文件）之间的区别，**Java** 介于两者之间。**Java** 程序要经过编译，但编译的结果不是机器码，而是 **Java** 字节码。在结构上 **Java** 字节码与机器码很相似，但用在虚构的计算机即 **Java** 虚拟机（**JVM**）上。执行编译后的 **Java** 程序的计算机 模拟 **JVM** 解释 **Java** 字节码。**Java** 程序可在不同机器上的不同图形操作系统上运行，所以是具有 平台独立性的程序。

虽然本书着重讲了用电信号在线路上传输信号和信息，但一种更有效的方式是通过光纤

——由玻璃或聚合体制造的小管道，可从不同角度传输光信号——来传输光信号。通过光纤传输光信号可以达到以吉赫计算的数据传输速率——即每秒几百万位。

所以，似乎是光子而不是电将要负责未来家庭和办公室的大量信息传输，它将比摩尔斯 电码的点划更快，也比那些我们曾用来午夜与好朋友通信而精心设计的闪灯更快。

Table of Contents

[第1章 电筒密谈](#)

[第2章 编码与组合](#)

[第3章 布莱叶盲文与二元编码](#)

[第4章 手电筒剖析](#)

[第5章 绕过拐弯的通信](#)

[第6章 发报机与断电器](#)

[第7章 十进制记数法](#)

[第8章 其他进位制记数法](#)

[第9章 二进制数](#)

[第10章 逻辑与开关](#)

[第11章 逻辑门电路](#)

[第12章 二进制加法机](#)

[0 1 1 0 0 1 0 1](#)

[1 1 1 1](#)

[第13章 如何实现减法](#)

[第14章 反馈与触发器](#)

[第15章 字节与十六进制](#)

[第16章 存储器组织](#)

第17章 自动操作

第18章 从算盘到芯片

第19章 两种典型的微处理器

第20章 ASCII码和字符映射

第21章 总线连接

第22章 操作系统

第23章 定点数和浮点数

第24章 高级语言和低级语言

第25章 图形化革命